JavaGI: The Interaction of Type Classes with Interfaces and Inheritance

Stefan Wehr
factis research GmbH, Freiburg, Germany
and
Peter Thiemann
University of Freiburg, Germany

The language JavaGI extends Java 1.5 conservatively by a generalized interface mechanism. The generalization subsumes retroactive and type-conditional interface implementations, binary methods, symmetric multiple dispatch, interfaces over families of types, and static interface methods. These features make certain coding patterns redundant, increase the expressiveness of the type system, and permit solutions to extension and integration problems with components in binary form, for which previously several unrelated extensions had been suggested.

This article explains JavaGI and motivates its design. Moreover, it formalizes a core calculus for JavaGI and proves type soundness, decidability of typechecking, and determinacy of evaluation. The article also presents the implementation of a JavaGI compiler and an accompanying runtime system. The compiler, based on the Eclipse Compiler for Java, offers mostly modular static typechecking and fully modular code generation. It defers certain well-formedness checks until load time to increase flexibility and to enable full support for dynamic loading. Benchmarks show that the code generated by the compiler offers good performance. Several case studies demonstrate the practical utility of the language and its implementation.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: JavaGI, Java, generalized interfaces, retroactive interface implementations, external methods, open classes, multimethods, multiple dispatch, type conditionals, multi-headed interfaces, explicit implementing types, binary methods, static interface methods, implementation constraints, constraint entailment, subtyping, formalization, implementation, case studies, benchmarks

1. INTRODUCTION

Extending existing software, adapting it to new requirements, and integrating software from different sources into a new product are frequent activities of software

This work was done while Stefan Wehr was a PhD student at the University of Freiburg.
Authors' addresses: Stefan Wehr, factis research GmbH, Merzhauserstraße 177, 79100 Freiburg, Germany; Peter Thiemann, University of Freiburg, Georges-Köhler-Allee 079, 79110 Freiburg, Germany.

developers. Object-oriented languages such as Java or C# provide at least two features that are helpful for dealing with these challenges: interfaces and inheritance.

The interface mechanism is particularly attractive because interfaces separate specification from implementation and support multiple inheritance without the semantic problems caused by diamond inheritance [Sakkinen 1989]. However, interfaces and inheritance often do not suffice for solving various extension, adaptation, and integration problems, as documented by numerous proposals [Harrison and Ossher 1993; Kiczales et al. 1997; Wadler 1998; Mezini and Ostermann 2002; Torgersen 2004; Odersky and Zenger 2005b; Clifton et al. 2006; Warth et al. 2006]. These problems become worse if libraries and other components are only available in binary form or cannot be modified for other reasons.

Interestingly, Lämmel and Ostermann [2006] demonstrated that type classes [Wadler and Blott 1989], a structuring mechanism related to object-oriented–style interfaces but introduced by the functional programming language Haskell [Peyton Jones 2003], provide clean solutions to a number of software extension and integration problems. Their findings raise the question whether object-oriented–style interfaces could give rise to similar solutions if extended and generalized in the direction of type classes. A related question is whether such an extension could raise the expressiveness of the type system to prevent tedious coding patterns, unsafe cast operations, run-time exceptions, and code duplication. After all, many examples demonstrate that Haskell's type system provides powerful abstractions and strong static guarantees through type classes [Kiselyov et al. 2004; Jones 1993; Peyton Jones et al. 1997; Kiselyov and Shan 2004].

This paper answers that questions affirmatively by presenting JavaGI [Wehr et al. 2007; Wehr and Thiemann 2008; 2009a; 2009b; Wehr 2010], a conservative (i.e. backwards-compatible) extension of Java[1] that generalizes Java's interface concept with features from Haskell type classes.[2] More specifically, JavaGI generalizes Java interfaces in the following dimensions:

—*Retroactive Interface Implementations.* The declaration that a class implements an interface may be separate from the definition of the interface and the implementing class. Both the interface and the implementation relationship may be declared after the class and without touching it.

—*Explicit Implementing Types.* An interface may explicitly reference its implementing type, which enables the declaration of binary methods [Bruce et al. 1995].

—*Multi-Headed Interfaces.* An interface may be multi-headed: It may span multiple types to specify mutual dependencies.

—*Symmetric Multiple Dispatch.* Interface methods depending on implementing

[1]Throughout this article, the term "Java" always refers to version 1.5 of the Java programming language [Gosling et al. 2005].

[2]Some of the results presented here are based on previous work [Wehr et al. 2007; Wehr and Thiemann 2009a]. This article extends that work with a complete formalization, including the specification of a dynamic semantics and all theorems and proofs. Moreover, it contains an extensive discussion of the design and puts the work in a larger perspective with a comprehensive survey of related work. There is some further discussion of the differences at the beginning of Section 7.

types in argument positions (binary methods and certain methods in multi-headed interfaces) are subject to symmetric multiple dispatch [Chambers 1992].

—*Implementation Constraints.* An interface may not only be used as a type but also in a constraint to restrict a type or a family of types.

—*Type Conditionals.* Methods and retroactive interface implementations may depend on type constraints, which enables type-conditional methods and interface implementations.

—*Static Interface Methods.* An interface may contain static methods.

These features increase the expressiveness of the language and permit solutions to extension, adaptation, and integration problems with components in binary form for which unrelated extensions had been suggested before. Compared with other work, retroactive interface implementations permit non-invasive and in-place object adaption [Warth et al. 2006], supersede the Adapter and Visitor patterns [Gamma et al. 1995], and enable a solution to (a restricted version of) the expression problem [Wadler 1998; Torgersen 2004]; explicit implementing types are related to work on `MyType` and `ThisType` [Bruce et al. 1997; Bruce and Foster 2004] and supersede (and simplify) certain instances of F-bounded polymorphism [Canning et al. 1989]; multi-headed interfaces provide a restricted form of family polymorphism [Ernst 2001]; symmetric multiple dispatch supersedes the double dispatch pattern [Ingalls 1986]; implementation constraints avoid certain cast operations; type conditionals avoid code duplication or run-time errors [Huang et al. 2007]; and static interface methods supersede uses of the Factory pattern [Gamma et al. 1995].

Contributions and Overview

This article presents the design, a formalization, and an implementation of JavaGI:

—It introduces the features of JavaGI, highlights the underlying design principles, and explains the JavaGI-specific extensions of Java's type system and execution model informally (Sections 2 and 3). The design of JavaGI is unique in that it avoids a patchwork of unrelated features but offers a unifying conceptual view that addresses several seemingly disparate concerns.

—It formalizes a core calculus of JavaGI in the style of Featherweight Generic Java [Igarashi et al. 2001] and proves type soundness, decidability of typechecking, and determinacy of evaluation (Section 4). The main technical challenges of the formalization lie in the combination of type classes with subtyping and in the definition of a decidable subtyping and typing algorithm.

—It reports on an implementation of a JavaGI compiler and an accompanying run-time system with support for mostly modular typechecking, fully modular compilation, and dynamic loading (Section 5).[3]

—It summarizes the outcome of a number of case studies and describes the results of several performance benchmarks to demonstrate the practical utility of JavaGI and its implementation (Section 6).

—It puts JavaGI in perspective by providing a comprehensive survey and discussion of related work in Section 7.

[3]The implementation is available on the web [Wehr 2009].

```
abstract class Expr {
  abstract int eval();
}
class IntLit extends Expr {
  int value;
  IntLit(int value) {
    this.value = value;
  }
  int eval() {
    return this.value;
  }
}
```

```
class PlusExpr extends Expr {
  Expr left;
  Expr right;
  PlusExpr(Expr left, Expr right) {
    this.left = left;
    this.right = right;
  }
  int eval() {
    return this.left.eval() + this.right.eval();
  }
}
```

Figure 1: Type hierarchy for expressions.

Section 8 concludes and gives pointers to future work.

2. FEATURES OF JAVAGI

This section introduces the features of JavaGI through a series of examples, which also demonstrate how JavaGI solves the programming problems mentioned in the introduction (Section 2.1–2.7). The section closes by comparing the solutions in JavaGI with corresponding solutions in Java (Section 2.8) and by highlighting the underlying design principles of JavaGI (Section 2.9).

The examples are all based on the simple type hierarchy for expressions shown in Figure 1. We assume that it is not possible to modify the source code of the expression hierarchy. As JavaGI is an extension of Java, JavaGI code (and Java code where appropriate) refers to common classes and interfaces from the Java API [Sun Microsystems 2004b].[4] See Appendix A for a definition of JavaGI's syntax.

2.1 Retroactive Interface Implementations

The expression hierarchy in Figure 1 supports only evaluation of expressions. Now suppose that we also want to produce nicely formatted string output from expression instances. To implement this functionality, we would like to use a library such as *The Java Pretty Printer Library*[5]. This library provides an interface that classes with pretty-printing support must implement:[6]

```
interface PrettyPrintable { String prettyPrint(); }
```

A Java programmer cannot add an implementation for the PrettyPrintable interface to the classes of the expression hierarchy because we assumed earlier that the source code of these classes is unmodifiable. Instead, a Java programmer would presumably use the Adapter pattern [Gamma et al. 1995] and create a parallel hierarchy of expression adapters complying to the PrettyPrintable interface (see Section 2.8.1 for further discussion).

[4]The code uses classes and interfaces from the packages java.lang, java.util, and java.io without further qualification.
[5]`http://jpplib.sourceforge.net/`
[6]We slightly modified the interface for the purpose of presentation.

A JavaGI solution can avoid the Adapter pattern because JavaGI supports *retroactive interface implementations* where the implementation of an interface may be separate from the implementing class and can thus be provided afterwards. Here are three *implementation definitions* for the PrettyPrintable interface with the classes Expr, IntLit, and PlusExpr acting as the *implementing types* (enclosed in square brackets '[...]'):

```
implementation PrettyPrintable [Expr] {
  abstract String prettyPrint();
}
implementation PrettyPrintable [IntLit] {
  String prettyPrint() { return String.valueOf(this.value); }
}
implementation PrettyPrintable [PlusExpr] {
  String prettyPrint() {
    return "(" + this.left.prettyPrint() + " + " + this.right.prettyPrint() + ")";
  }
}
```

The prettyPrint method for the abstract base class Expr remains abstract because there is no sensible default implementation, but it is nevertheless required to define it. JavaGI guarantees that the implementation of prettyPrint is *complete*: there exists a non-abstract definition of prettyPrint for each concrete subclass of Expr. The JavaGI compiler performs a global check to ensure this completeness property, relying on an incremental load-time check to cater for subsequently added or modified classes (see Sections 3.4.6 and 3.4.7).

In the body of the two other prettyPrint methods, the static type of **this** is the implementing type of the surrounding implementation definition. That is, in the implementation for IntLit, **this** has static type IntLit, so the field access **this**.value is type correct. Similarly, in the implementation for PlusExpr, **this** has type PlusExpr, so the field accesses **this**.left and **this**.right are valid. We can invoke prettyPrint recursively on these fields because there is an implementation of PrettyPrintable for Expr, albeit an abstract one.

Methods of retroactive interface implementations are subject to dynamic dispatch, just as ordinary interface and class methods.[7] For instance, the recursive call **this**.left.prettyPrint() in the implementation for PlusExpr selects the method to invoke based on the dynamic type of the receiver **this**.left. Hence, the call

```
new PlusExpr(new IntLit(1), new IntLit(2)).prettyPrint()
```

correctly returns `"(1 + 2)"`.

The implementations of PrettyPrintable for Expr, IntLit, and PlusExpr not only add the prettyPrint method to these classes but also make them compatible with the interface type PrettyPrintable. For example, we may pass an object of type PlusExpr to a method expecting an object of type PrettyPrintable:

```
class SomePrinter {
  void print(PrettyPrintable pp) {
```

[7] In contrast, extension methods in C# 3.0 [ECMA International 2005] are subject to static dispatch.

```
    String s = pp.prettyPrint(); System.out.println(s);
  }
  void usePrint() {
    PlusExpr expr = new PlusExpr(new IntLit(1), new IntLit(2));
    print(expr); // use a "PlusExpr" instance at type "PrettyPrintable"
  }
}
```

Retroactive implementation definitions can be placed in arbitrary compilation units. For example, it is possible to place the three implementations shown earlier in three different compilation units, all of which may be different from the compilation units of the expression hierarchy and the PrettyPrintable interface.

This flexibility together with dynamic dispatch on retroactively implemented methods implies extensibility in the operation dimension and thus eliminates the need for the Visitor pattern [Gamma et al. 1995]: to add a new operation, simply define an interface for the operation and provide suitable implementation definitions. Extensibility in the data dimension is also straightforward: add a new subclass of Expr and provide interface implementations for existing operations, unless the default for the base class suffices. Hence, JavaGI permits a simple and elegant solution to (a restricted version of) the expression problem [Wadler 1998; Torgersen 2004] (see Section 7.4.1 for further discussion).

JavaGI does not require explicit import statements for retroactive implementation definitions. Instead, all retroactive implementations presented to the JavaGI compiler are automatically in scope. Imposing stricter visibility rules at compile time is not necessary because JavaGI's run-time system puts all implementation definitions into a global pool anyway as explained in Section 5.3.

2.2 Explicit Implementing Types

A *binary method* [Bruce et al. 1995] is a method requiring the receiver type and some of the argument types to coincide. According to Bracha [2004], the definition of a binary method in Java requires F-bounded polymorphism [Canning et al. 1989] and possibly wildcards [Torgersen et al. 2004] (see Section 2.8.2 for further discussion). In contrast, JavaGI directly supports binary methods in interfaces through *explicit implementing types*. The following interface defines an equality operation that restricts comparison to objects with compatible types.

```
interface EQ { boolean eq(This 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. Hence, eq qualifies as a binary method. The next example uses eq to define a generic function that obtains the position of a specific element in a list.

```
class Lists {
  static <X implements EQ> int pos(X x, List<X> list) {
    int i=0; for (X y : list) { if (x.eq(y)) return i; else i++; }
    return −1;
  }
}
```

We specify that X has to implement the EQ interface through the *implementation constraint* X **implements** EQ.

When typechecking an implementation of EQ, the JavaGI compiler replaces the type variable **This** with the concrete implementing type. Here are EQ implementations for the classes of the expression hierarchy from Figure 1:

```
implementation EQ [Expr] {
  boolean eq(Expr that) { return false; }
}
implementation EQ [IntLit] {
  boolean eq(IntLit that) { return this.value == that.value; }
}
implementation EQ [PlusExpr] {
  boolean eq(PlusExpr that) { return this.left.eq(that.left) && this.right.eq(that.right); }
}
```

Given variables e, le, i, and li with static types Expr, List<Expr>, IntLit, and List<IntLit>, respectively, the following invocations of Lists.pos typecheck successfully:

```
Lists.pos(e, le); Lists.pos(i, le); Lists.pos(i, li);
```

Interface methods with explicit implementing types (such as eq) are similar to multimethods [Chambers 1992], with implementation definitions providing concrete specializations. Thus, dynamic method resolution selects the most specific implementation dynamically by extending dynamic dispatch to all parameters declared as implementing types (symmetric multiple dispatch, discussed in Section 7.5). Hence, invocations of eq dispatch on both the receiver and the first argument of the call.

Let us explain this behavior by considering the following variable declarations:

```
Expr plus1 = new PlusExpr(new IntLit(1), new IntLit(2));
Expr plus2 = new PlusExpr(new IntLit(1), new IntLit(2));
Expr intLit = new IntLit(42);
```

All three variables have static type Expr. Nevertheless, the call plus1.eq(plus2) invokes the eq method of the implementation for PlusExpr because both the receiver plus1 and the argument plus2 have dynamic type PlusExpr. On the other hand, the call plus1.eq(intLit) invokes the eq method as implemented for the base class Expr because dynamic dispatch on the argument intLit rules out eq for PlusExpr and dynamic dispatch on the receiver plus1 rules out eq for IntLit.

2.3 Type Conditionals

If the elements of two lists are comparable, then the lists should be comparable, too. JavaGI can express this implication with a *type-conditional interface implementation* [Huang et al. 2007; Litvinov 1998].

```
implementation<X> EQ [List<X>] where X implements EQ {
  boolean eq(List<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();
      if (!thisX.eq(thatX)) return false;
```

```
    }
    return !(thisIt.hasNext() || thatIt.hasNext());
  }
}
```

The implementation of EQ for List<X> is parameterized over X, the type of list elements. The constraint X **implements** EQ makes the eq operation available on objects of type X and ensures that only lists with comparable elements implement EQ. For example, if l1 and l2 have type List<Expr> and l3 has type List<List<Expr>>, then the method calls l1.eq(l2) and Lists.pos(l1, l3) are both valid.

The notation **where** ..., reminiscent of .NET generics [Kennedy and Syme 2001; Yu et al. 2004], is not only available for constraints on interface implementations, but also for constraints on ordinary classes and interfaces. It may even be used to constrain type parameters of a class or interface on the basis of individual methods, as the next example shows.

```
class Box<X> {
  X x;
  boolean containedBy(List<X> list) where X implements EQ {
    return Lists.pos(this.x, list) >= 0;
  }
}
```

The class Box itself places no constraint on its type parameter X. Thus, it may be instantiated with arbitrary types. However, method containedBy is only available if the actual type argument implements EQ; in other words, containedBy is a *type-conditional method* [Emir et al. 2006; Kennedy and Russo 2005]. For instance, an invocation of containedBy on a value of type Box<Expr> is valid, whereas an invocation on a value of type Box<String> is rejected by the compiler (unless there is an implementation of EQ for String).

2.4 Static Interface Methods

In addition to operating on expressions, we also want to parse them from a string representation. As there are many situations (e.g., XML deserialization, parsing of XPath expressions) where we need to create an object from an external string representation, we would like to abstract over these different situations.

As an example, consider a generic line processor: a method that loops over the lines of a given input stream, parses them, and then passes the result to some consumer. To reuse the code of looping over the input stream, we need to abstract over the parser and the consumer. Abstracting over the consumer is easily done using a plain Java interface:

```
// Consumes values of type X
interface Consumer<X> { void consume(X x); }
```

However, a similar solution does not work for parsing because a parser acts like an additional class constructor: it creates an object from a string representation, so the parse method cannot be in an instance method of the object being parsed. In this situation, Java programmers routinely use the Factory pattern [Gamma et al. 1995] (cf. Section 2.8.4). In JavaGI, programmers may abstract over such "constructor-like" methods using static interface methods:

```
// Parses a string and returns a value of the implementing type
interface Parseable { static This parse(String s); }
```

(Again, the result type **This** refers to the implementing type.) This interface makes it easy to implement the line processor:

```
class LineProcessor {
  static <X> void process(InputStream in, Consumer<X> c) throws IOException
    where X implements Parseable {
    BufferedReader br = new BufferedReader(new InputStreamReader(in));
    String line;
    while ((line = br.readLine()) != null) {
      X x = Parseable[X].parse(line); // parse the line ...
      c.consume(x); // ... and consume it
    }
  }
}
```

The expression Parseable[X].parse(s) invokes the parse method of Parseable with X as the implementing type. The invocation is well-typed because we require the constraint X **implements** Parseable (see Section 3.1). It returns an object of type X, which we pass to the consume method.

Given an implementation of Parseable for Expr, we can use the line processor to implement a simple Read-Evaluate-Print-Loop:

```
class REPL {
  public static void main(String... args) throws IOException {
    LineProcessor.process(System.in, new Consumer<Expr>() {
      public void consume(Expr e) {
        System.out.println(e.prettyPrint() + " => " + e.eval());
      }
    });
  }
}
```

2.5 Implementation Inheritance

Suppose we would like to have a richer set of operations available for the expression hierarchy, as expressed by the following interface:

```
interface RichExpr {
  int depth(); // Computes the depth of the expression
  int size(); // Computes the size of the expression
  List<RichExpr> subExprs(); // Returns all direct sub−expressions
}
```

Providing direct implementations of depth and size for Expr and its subclasses would duplicate work because both methods can be implemented in terms of the subExprs method. A Java programmer can only avoid this sort of code duplication proactively by extending the abstract Expr class with methods depth, size, and subExprs where the former two refer to the latter, which is only provided as a default implementation that always returns an empty list. Furthermore, the subclasses of Expr would have to override this default implementation as needed. In our

example, only the class PlusExpr would be concerned. None of these manipulations are possible without the source code of Expr and all subclasses that have a non-empty list of subexpressions.

Alternatively, the programmer could extend all the leaf subclasses of Expr and create a parallel class hierarchy RichExpr mirroring the original Expr class hierarchy. However, this approach would not be type compatible with the original program.

JavaGI offers *abstract implementation definitions* as a more flexible way for writing (partial) default implementations. In this case, an abstract implementation of RichExpr with RichExpr as the implementing type reflects the intention of implementing some methods of RichExpr in terms of other methods of RichExpr. Here is the code for the partial default implementation of RichExpr:[8]

```
abstract implementation RichExpr [RichExpr] {
  int depth() {
    int i = 0; for (RichExpr e : subExprs()) { i = Math.max(i, e.depth()); }
    return i+1;
  }
  int size() { ... }
}
```

Other implementations of RichExpr may then inherit from this abstract implementation:

```
implementation RichExpr [Expr] extends RichExpr [RichExpr] {
  List<RichExpr> subExprs() { return new LinkedList<RichExpr>(); }
}
```

We use the syntax "**extends** RichExpr [RichExpr]" to specify the super implementation. The effect of the **extends** clause is pure code reuse: RichExpr [Expr] inherits the definitions of depth and size from RichExpr [RichExpr].[9]

An implementation for PlusExpr completes the example:

```
implementation RichExpr [PlusExpr] extends RichExpr [Expr] {
                          // alternative extends clause: ''extends RichExpr [RichExpr]''
  List<RichExpr> subExprs() {
    List<RichExpr> list = new LinkedList<RichExpr>();
    list.add(this.left); list.add(this.right);
    return list;
  }
}
```

These examples refer to a super implementation by explicitly stating the interface and the implementing type. Alternatively, we may provide explicit names for implementations [Kahl and Scheffczyk 2001] and then use these names in the **extends** clause. In this case, the three implementations of RichExpr look as follows:

[8]Abstract implementation definitions and implementation definitions with abstract methods (which are not necessarily abstract as a whole) are two different things. The former do not introduce a new subtyping relationship between the implementing type and the interface, whereas the latter do. Hence, JavaGI's type system treats abstract implementations more liberally and imposes fewer restrictions on them (cf. Section 3.4).

[9]The notation "I [T]" denotes the retroactive implementation of interface I for type T.

```
abstract implementation RichExpr [RichExpr] as DefaultImpl { ... }
implementation RichExpr [Expr] as ExprImpl extends DefaultImpl { ... }
implementation RichExpr [PlusExpr] extends ExprImpl { ... }
```

2.6 Dynamic Loading of Retroactive Implementations

JavaGI's retroactive interface implementations integrate nicely with the dynamic loading capabilities of Java. Here is code that loads an (imaginary) subclass MultExpr of Expr together with its retroactive implementation of the PrettyPrintable interface. The code then constructs a new instance of MultExpr (we expect the class to have a constructor taking two Expr arguments) and invokes the prettyPrint method on the new instance.

```
Class<?> clazz = javagi.runtime.RT.classForName("MultExpr", PrettyPrintable.class);
Expr e = (Expr) clazz.getDeclaredConstructor(Expr.class, Expr.class)
                     .newInstance(new IntLit(2), new IntLit(21));
String s = e.prettyPrint();
System.out.println(s);
```

The method classForName(String name, Class<?>... ifaces), provided by the runtime system of JavaGI, simultaneously loads a class and its implementations of all interfaces given. In the example just shown, it is not possible to load MultExpr first and the PrettyPrintable implementation at some later point. This approach would permit the invocation of the prettyPrint method on a MultExpr object without loading the PrettyPrintable implementation at all. Such an invocation would lead to a run-time error because the only applicable prettyPrintable method would be the abstract version in the implementation of PrettyPrintable for Expr. Consequently, JavaGI's completeness check for abstract methods would prevent MultExpr from being loaded in the first place. Loading MultExpr and its PrettyPrintable implementation simultaneously avoids the problem.

2.7 Multi-Headed Interfaces

So far, we only considered interfaces with exactly one implementing type. However, we can easily generalize the interface concept to include *multi-headed interfaces*. Such interfaces relate multiple implementing types and their methods and thus can place mutual requirements on the methods of all participating types. For instance, here is a multi-headed interface for the Observer pattern [Gamma et al. 1995]:[10]

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

[10]Two parties participate in the Observer pattern: subject and observer. Every observer registers itself with one or more subjects. Whenever a subject changes its state, it notifies its observers by sending itself for scrutiny.

A multi-headed interface names the implementing types (Subject and Observer in this case) explicitly through type variables enclosed in square brackets '[...]'. Moreover, it groups methods by receiver type. In the example, the ObserverPattern interface demands that the Subject part provides the methods register and notifyObservers, whereas the Observer part has to provide an update method.

Implementations of multi-headed interfaces are defined analogously to implementations of single-headed interfaces.[11] Assume that there are classes ExprPool, which maintains a pool of expressions scheduled for evaluation, and ResultDisplay, which displays the result of evaluating an expression on the screen.

```
class ExprPool {
  void register(ResultDisplay d) { ... } void notifyObservers() { ... } ...
}
class ResultDisplay { ... }
```

Class ResultDisplay is an observer for ExprPool: whenever ExprPool evaluates an expression, it notifies ResultDisplay to update the screen. We can make this relationship explicit by providing a corresponding implementation of the ObserverPattern interface:

```
implementation ObserverPattern [ExprPool, ResultDisplay] {
  /* No need to specify methods for receiver ExprPool because
     this class already contains the required methods. */
  receiver ResultDisplay {
    void update(ExprPool m) { ... }
  }
}
```

In conjunction with multi-headed interfaces, JavaGI's constraint notation is particularly useful because it allows programmers to constrain multiple types. The following example uses this mechanism to demand that the type variables S and O together implement the ObserverPattern interface:[12]

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

Because ExprPool and ResultDisplay implement ObserverPattern, the method invocation genericUpdate(**new** ExprPool(), **new** ResultDisplay()) is type correct.

Similar to single-headed interfaces, methods of multi-headed interfaces also preserve dynamic dispatch. As with binary methods, JavaGI takes an approach similar to multimethods and dispatches on the receiver as well as on all parameters declared as implementing types (symmetric multiple dispatch, cf. Section 7.5). However, unlike their single-headed counterparts, multi-headed interfaces cannot be used as proper types because their implementation usually involves more than one type.

[11] Single-headed interfaces are interfaces with exactly one implementing type. In general, the term "n-headed interface" refers to an interface with n implementing types.

[12] The first version of JavaGI [Wehr et al. 2007] used the notation [S,O] **implements** ObserverPattern instead of S*O **implements** ObserverPattern. We opted for the new notation to simplify parsing of JavaGI code.

We end the discussion of multi-headed interfaces by remarking that the notation for single-headed interfaces used so far is just syntactic sugar. Internally, a single-headed interface is represented in the same way as a multi-headed interface. For example, the EQ interface from Section 2.2 is fully spelled out as:

```
interface EQ [This] {
  receiver This { boolean eq(This that); }
}
```

2.8 Comparison with Java

The preceding subsections introduced the main features of JavaGI and demonstrated how these features solve several important programming problems. In the following, we compare the JavaGI solutions with corresponding solutions in plain Java.

2.8.1 *Retroactive Interface Implementations.* As already noted in Section 2.1, Java does not offer the possibility of implementing interfaces such as PrettyPrintable without changing the classes of the expression hierarchy in Figure 1. As a workaround, Java programmers often use the Adapter pattern [Gamma et al. 1995; Hummel and Atkinson 2009]. Applying this design pattern to the problem in Section 2.1 requires adapter classes for each concrete subclass of Expr and a factory class that adapts expressions according to their run-time type.

Assessment. The Adapter pattern has some disadvantages compared with JavaGI's retroactive implementations:

—It requires explicit conversion between the original and the adapted object.
—It causes object schizophrenia [Sekharaiah and Ram 2002; Hölzle 1993]: an expression and its adapted form are no longer identical according to the == operator.
—It hides the original interface of the object being adapted. Gamma et al. [1995] suggest *two-way adapters* as a potential solution to this problem.
—It requires a factory class for constructing adapter objects. Adding new expression forms requires changes to this factory class.
—It has the tendency to "infect" large areas of a program. For example, treating a list of expressions as a list of pretty-printable objects requires an adapter for the list [Hölzle 1993]. (The list adapter adapts the individual elements whenever they are retrieved from the list.)

2.8.2 *Explicit Implementing Types.* Section 2.2 demonstrates that JavaGI specifies signatures for binary methods through explicit implementing types. That section also argues that the specification of a binary method signature in Java requires F-bounded polymorphism and possibly wildcards. Figure 2 re-implements the example from Section 2.2 in Java to substantiate this claim. Bracha [2004] gives a different example for the same purpose.

Given variables le, li, e, and i with static types List<EQExpr>, List<EQIntLit>, EQExpr, and EQIntLit, the invocations Lists.pos(e, le) and Lists.pos(i, le) typecheck. However, in contrast to the JavaGI solution, the invocation Lists.pos(i, li) does not typecheck because it causes the type parameter X to be instantiated with EQIntLit but EQIntLit is not a subtype of EQ<EQIntLit> (but of EQ<EQExpr>).

```
// Java
interface EQ<X> { boolean eq(X that); }
class Lists {
  static <X extends EQ<X>> int pos(X x, List<X> list) { ... }
}
abstract class EQExpr implements EQ<EQExpr> {
  public boolean eq(EQExpr that) { return false; } // eval removed for simplicity
}
class EQIntLit extends EQExpr {
  int value; EQIntLit(int value) { this.value = value; }
  public boolean eq(EQExpr that) { // simulate multiple dispatch
    if (that instanceof EQIntLit) return this.value == ((EQIntLit) that).value;
    else return super.eq(that);
  }
}
class EQPlusExpr extends EQExpr { /* code omitted for brevity */ }
```

Figure 2: Binary methods in plain Java. The code avoids the problem of implementing EQ retroactively for Expr and its subclasses by defining a variant of the expression hierarchy from Figure 1 that directly implements Java's version of EQ.

Such flexibility requires an improved version of pos's signature with wildcards:

```
// Java code
static <X extends EQ<? super X>> int betterPos(X x, List<X> list) { /* code as before */ }
```

The bound EQ<? super X> states that X does not need to be a subtype of EQ<X>; instead, it only has to be a subtype of EQ<T> where T is some arbitrary supertype of X. With the improved version of pos, the invocation betterPos(i, li) typechecks successfully because EQIntLit is a subtype of EQ<EQExpr> and EQExpr is a supertype of EQIntLit. (The invocations betterPos(e, le) and betterPos(i, le) typecheck too).

Assessment. Comparing the JavaGI version with its Java counterpart reveals that explicit implementing types are syntactically much simpler than F-bounds and wildcards. Moreover, JavaGI provides multiple dispatch on explicit implementing types, something that the Java approach has to simulate by hand (e.g., by **instanceof** tests as in the implementation of eq in class EQIntLit).

On the other hand, the solution in JavaGI only works in combination with interfaces whereas Java's solution also works in a setting without interfaces. Further, Java's approach is somewhat more flexible; for example, a class C may implement EQ<T> for some arbitrary type T, which may be totally unrelated to C. However, it is unclear whether this greater flexibility is really needed in practice.

2.8.3 *Type Conditionals.* Java neither supports type-conditional interface implementations nor type conditions on methods restricting type parameters other than that of the method itself. A common approach to simulate these features is checking the type conditions not statically but dynamically through run-time casts. A different approach omits the type-conditional parts from the base class but creates a new subclass which then places the type conditions on its generic arguments.

Both approaches have disadvantages compared with the JavaGI solution presented

in Section 2.3: the first approach may lead to unexpected run-time errors, whereas the second approach requires boilerplate code to be written and does not offer much flexibility because the type-conditional parts are not available for the base class even if its type parameters meet the type conditions. Even worse, the boilerplate code grows exponentially in the number of independent type conditions because each combination of type conditions gives rise to a new subclass. To see this, consider a class with k type conditions, each of which enables a method f_i. To support all combinations of type conditions, 2^k subclasses are required where each subclass supports some subset of the methods.

2.8.4 *Static Interface Methods.* JavaGI programmers abstract over constructor-like methods with static interface methods. In Java, programmers rely on the Factory pattern [Gamma et al. 1995] instead. Implementing the line processor from Section 2.4 with the Factory pattern requires an interface

```
interface Parser<X> { X parse(String s); }
```

and the following modified signature of method process in class LineProcessor:

```
static <X> void process(InputStream in, Consumer<X> c, Parser<X> p) throws IOException
```

The extra parameter p simulates the JavaGI constraint X **implements** Parseable of the corresponding signature in Section 2.4. While JavaGI implicitly passes evidence for this constraint, a Java programmer has to supply the extra parameter explicitly. For the tiny example from Section 2.4, the extra parameter does not make a big difference, but explicitly maintaining it over a long sequence of method calls quickly becomes a burden.

2.8.5 *Multi-Headed Interfaces.* JavaGI's multi-headed interfaces specify mutual dependencies between several types. In the literature, this phenomenon is called *family polymorphism* [Ernst 2001]. It is well known [Ernst 2001] that object-oriented languages such as Java do not support family polymorphism in a statically safe and flexible way. JavaGI, however, provides a type-safe and sufficiently expressive form of family polymorphism, as demonstrated by the example in Section 2.7 (see also Section 7.3). In addition to family polymorphism, JavaGI's multi-headed interfaces in combination with explicit implementing types also support symmetric multiple dispatch, a feature not present in Java either.

2.9 Design Principles

We conclude the introduction of JavaGI by explaining the main design principles of the language.

Conservativeness. JavaGI is a conservative extension of Java: A program that works in Java works the same way in JavaGI. The JavaGI compiler translates all input programs to standard Java byte code [Lindholm and Yellin 1999], retaining the semantics and the performance characteristics of Java programs even in the presence of retroactive implementations. Conservativeness enables easy migration from Java to JavaGI and ensures full compatibility with existing Java APIs.

Extensibility. JavaGI imposes no restrictions on the placement of retroactive interface implementations. Implementation definitions can be placed in arbitrary

compilation units and arbitrary libraries. Extensibility maximizes flexibility and allows for a high degree of interworking between Java and JavaGI code.

Dynamicity. JavaGI fully supports dynamic loading. Not only classes and interfaces but also retroactive implementation definitions can be loaded dynamically at any time. Dynamicity ensures compatibility with existing Java libraries and frameworks. For example, dynamic loading is required to run JavaGI programs inside a servlet container.

Type Safety. JavaGI favors static type safety over dynamic checks. The language provides an expressive type system and checks as many properties as possible at compile time. It resorts to dynamic checks only if required to support extensibility or dynamicity. Static type safety prevents a whole class of software defects right from the start.

Modularity. JavaGI features fully modular compilation and mostly modular typechecking. Typechecking of a compilation unit does not need to access internals of other compilation units, and code generation processes each compilation unit in isolation. To support extensibility, dynamicity, and type safety at the same time, the JavaGI compiler abandons completely modular typechecking and performs certain global checks on the set of types and implementation definitions available. However, the compiler never assumes that it knows all implementation definitions (open-world assumption), so new implementations can be added at any time provided they do not conflict with existing ones. Further, the open-world assumption facilitates the extension of JavaGI libraries with new implementations without recompiling the libraries.

Transparency. JavaGI provides retroactive interface implementations in a transparent way. The run-time behavior of a retroactive implementation cannot be distinguished from that of a Java-style interface implementation, provided programs do not make use of Java's reflection API.[13] Furthermore, the compile-time characteristics of a retroactive and a Java-style implementation are very similar. Transparency enables programmers to reason about retroactive implementations in almost the same way as they reason about Java-style implementations.

3. TYPE CHECKING AND EXECUTING JAVAGI — AN INFORMAL ACCOUNT

This section investigates the JavaGI-specific extensions of Java's type system and execution model informally.

3.1 Constraint Entailment

Constraint entailment is a JavaGI-specific notion not present in Java's type system. It establishes the validity of constraints. JavaGI distinguishes two kinds of constraints, *subtype constraints* and *implementation constraints*.

— Subtype constraints generalize Java's type parameter bounds. Such a constraint has the form T **extends** U, where both T and U are types.[14] It is *valid* if T

[13]It is possible to devise a reflection API that provides uniform treatment of retroactive and Java-style interface implementations.

[14]Constraint declarations are restricted to the form X **extends** U, where X is a type variable. A Java type parameter bound X **extends** T_1&...&T_n is represented by multiple constraints

is a subtype of U (cf. Section 3.2).

— Implementation constraints have the form $\mathsf{T}_1{*}...{*}\mathsf{T}_n$ **implements** K where $\mathsf{T}_1, \ldots, \mathsf{T}_n$ are types and K is a n-headed interface. For simplicity, this informal discussion only considers the case where $n = 1$.

A constraint T **implements** K is valid in any of the following cases (see Section 4.2 for the complete list).

(1) T implements interface K in the Java sense: T is a class and T itself or a superclass of T has an explicit **implements** clause for K.
(2) T is a type variable declared to implement K or some of K's subinterfaces.
(3) A non-abstract retroactive implementation matches K and T (or some supertype of T unless K contains methods with the implementing type in result position). If the implementation is type conditional (see Section 2.3), then the constraints of the implementation must also be satisfied.

For instance, suppose a program contains the EQ implementations for Expr and List from Sections 2.2 and 2.3. The constraint LinkedList<Expr> **implements** EQ is valid by case (3):

—The implementing type of EQ does not appear in result position, so it is possible to lift LinkedList<Expr> to List<Expr>.

—There exists an implementation EQ [List<X>] (parameterized over X) that matches EQ and List<Expr> by instantiating X to Expr.

—The constraint of the implementation after instantiation is Expr **implements** EQ, which is valid because of the implementation EQ [Expr].

In contrast, LinkedList<String> **implements** EQ cannot by derived from the set of implementations defined in Sections 2.2 and 2.3 because String **implements** EQ does not hold.

An implementation constraint is stronger than an subtype constraint: validity of T **implements** K implies validity of T **extends** K, but the reverse implication is not always true. To demonstrate this fact, continue the example code from Section 2.2 and Section 2.3 as follows:

```
EQ e1 = new IntLit(42); // ok
EQ e2 = new LinkedList<Expr>(); // ok
if (e1.eq(e2)) ... // type error
```

While e1 and e2 can both be subsumed to the interface type EQ (see Section 3.2) and EQ **extends** EQ is clearly valid, the binary method call with e1 and e2 in the last line does not make sense as it would compare an integer with a list. For this reason, JavaGI requires EQ **implements** EQ to typecheck the call e1.eq(e2). But EQ **implements** EQ does not hold, so the JavaGI compiler correctly rejects the call.

Besides being stronger, implementation constraints may be used to constrain a group of types with a multi-headed interface, as demonstrated by the constraint S∗O **implements** ObserverPattern from Section 2.7. In contrast, a subtype constraint relates exactly two types. Furthermore, each invocation of a retroactively implemented or static interface method must eventually be sanctioned by a corresponding implementation constraint to ensure type soundness.

X **extends** $\mathsf{T}_1, \ldots,$ X **extends** T_n.

3.2 Subtyping

The subtyping relation, written $\mathsf{T} \leq \mathsf{U}$ for types T and U, indicates that an object of type T can also be used with type U. JavaGI's subtyping relation extends Java's: it considers more types to be subtypes of each other than Java.

To test whether $\mathsf{T} \leq \mathsf{U}$ holds, JavaGI first checks whether $\mathsf{T} \leq \mathsf{U}$ already holds in Java. Otherwise, $\mathsf{T} \leq \mathsf{U}$ can only hold if U is an interface type and T implements U. That is, there must be a supertype V of T (possibly T itself) such that the constraint V **implements** U holds.

With this approach, array types are no longer covariant with respect to subtyping if the two component types involved are subtypes in JavaGI but not in Java. That is, $\mathsf{T}[\,] \leq \mathsf{U}[\,]$ holds in JavaGI only if $\mathsf{T}[\,] \leq \mathsf{U}[\,]$ holds in Java. We do not consider this loss of covariant array types a major limitation because generic types are invariant in Java anyway and covariant array types are not statically type safe. Other researchers even proposed the removal of covariant array types from Java [Myers et al. 1997].

3.3 Method Typing

JavaGI's algorithm for typechecking a method invocation extends the corresponding algorithm employed by Java. If the rules of Java are sufficient to typecheck an invocation, then it also typechecks in JavaGI and the invocation is marked as a "Java call-site". Otherwise, JavaGI's constraint entailment tries to prove a suitable implementation constraint for the invocation.

In particular, assume that the method invocation not typeable in Java has the form $\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1,...,\mathsf{e}_n)$ for expressions $\mathsf{e}_0, \mathsf{e}_1, \ldots, \mathsf{e}_n$ with static types $\mathsf{T}_0, \mathsf{T}_1, \ldots, \mathsf{T}_n$. To typecheck the invocation, the JavaGI compiler first searches all interfaces accessible from the current compilation unit under their unqualified name for methods matching name m, receiver type T_0 and argument types $\mathsf{T}_1, \ldots, \mathsf{T}_n$. This process is very similar to the method typing algorithm described in sections 15.12.2 and 15.12.3 of *The Java Language Specification* [Gosling et al. 2005]. It includes inference of type arguments and it instantiates the implementing types of the current interface according to the signature of the method being examined and according to the types $\mathsf{T}_0, \ldots, \mathsf{T}_n$. If the compiler does not find any matching methods, typechecking fails.

Next, the compiler shrinks the resulting set of candidate methods by removing methods that are less specific than other candidate methods. If this process results in one candidate, typechecking succeeds and the compiler marks the invocation as a "JavaGI call-site". Otherwise, it rejects the method invocation as ambiguous.

There is a mechanism for resolving ambiguities by explicitly specifying which interface to search for candidate methods. For example, suppose that interface PrettyPrintable from Section 2.1 and another interface J are in scope. Furthermore, assume that J defines a method prettyPrint() and that Expr implements J. Then the call e.prettyPrint(), where e is a variable with static type Expr, is ambiguous. However, the syntax e.prettyPrint::PrettyPrintable() invokes the prettyPrint method of interface PrettyPrintable explicitly.

A static interface method invocation is always explicit. It includes the interface name and all implementing types to avoid potential ambiguities from the start.

3.4 Well-formedness Criteria for Programs

JavaGI's type system imposes certain global well-formedness criteria on the set of implementation definitions to guarantee that run-time lookup of retroactively implemented methods always finds a unique and most specific implementation definition that contains a non-abstract version of the method in question. 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 inherits its type-erasure semantics from Java [Bracha et al. 1998], so type arguments are not available when actually executing a program. Last but not least, the criteria establish decidability of constraint entailment and subtyping, and they enable efficient method lookup.

3.4.1 *Criterion: No Overlap.* 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.

For example, suppose a program contains the following two implementation definitions:

```
implementation PrettyPrintable [Box<IntLit>] { ... }
implementation PrettyPrintable [Box<PlusExp>] { ... }
```

Both implementation definitions would be candidates for an invocation such as `new Box<IntLit>(...).prettyPrint()` because the dispatch mechanism only looks at the erasure. The "no overlap" criterion rejects such a program.

3.4.2 *Criterion: 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. (The implementing types $\mathsf{T}_1, \ldots, \mathsf{T}_n$ and $\mathsf{U}_1, \ldots, \mathsf{U}_n$ of two implementations are *subtype compatible* if, and only if, for all $i \in \{1, \ldots, n\}$ either $\mathsf{T}_i \leq \mathsf{U}_i$ or $\mathsf{U}_i \leq \mathsf{T}_i$ holds.) 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 non-static 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*.)

The restriction on identical interface type arguments is necessary to avoid ambiguity in dynamic method lookup because JavaGI's type-erasure semantics maps different instantiations of an interface to the same run-time representation. Java also disallows multiple instantiation inheritance [Gosling et al. 2005, § 8.1.5].

A program containing two implementations of the same interface that have subtype-compatible implementing types but different non-dispatch types may also exhibit ambiguous method lookup at run time. For example, suppose that a program contains the ObserverPattern implementation for ExprPool and ResultDisplay from Section 2.7, as well as an ObserverPattern implementation for ExprPool and some class MyObserver. Then the call `new ExprPool().notify()` cannot be resolved unambiguously at run time because the two implementations differ only in the second implementing type (ResultDisplay and MyObserver), but it is not possible to determine this implementing type from the call `new ExprPool().notify()`. However, the second implementing type of ObserverPattern is a non-dispatch type (it is neither

the receiver nor an argument of notify), so the two ObserverPattern implementations considered violate the well-formedness criterion "unique non-dispatch types".

3.4.3 *Criterion: Downward Closed.* Any two non-abstract implementations of the same interface I must be downward closed. That is, if $\mathsf{T}_1, \ldots, \mathsf{T}_n$ and $\mathsf{U}_1, \ldots, \mathsf{U}_n$ are the implementing types of the two implementations given, and $\mathsf{V}_1, \ldots, \mathsf{V}_n$ is a vector of types such that each V_i is a maximal element of the set of lower bounds of T_i and U_i, then an implementation of interface I with implementing types $\mathsf{V}_1, \ldots, \mathsf{V}_n$ must exist.

This criterion rules out ambiguity of dynamic method lookup in cases like the following, where the chooseIntLit method is to return the IntLit instance among its arguments:

```
interface ChooseIntLit [Expr1, Expr2] { receiver Expr1 { IntLit chooseIntLit(Expr2 that); }}
implementation ChooseIntLit [Expr, IntLit] { ... }
implementation ChooseIntLit [IntLit, Expr] { ... }
```

The call **new** IntLit(42).chooseIntLit(**new** IntLit(3)) is ambiguous with these definitions because both implementations are applicable but none is more specific than the other. JavaGI rules out such programs because the two implementations are not downward closed. To make the program well-formed requires a third implementation with implementing types IntLit and IntLit.

The "downward closed" criterion also rules out ambiguities resulting from retroactive implementations with interfaces as implementing types in interaction with multiple interface inheritance [Wehr 2010, page 26].

3.4.4 *Criterion: Consistent Type Conditions.* Constraints on non-abstract implementations must be consistent with subtyping: if the implementing types of a non-abstract implementation $\mathcal{I}_1$ are pairwise subtypes of the implementing types of another non-abstract implementation $\mathcal{I}_2$, then the constraints of $\mathcal{I}_2$ must imply the constraints of $\mathcal{I}_1$.

Without this criterion, JavaGI would need run-time constraint entailment to rule out certain implementations when performing dynamic method lookup. For example, consider the following extension of code from Section 2.3:

```
// repeated for clarity
implementation<X> EQ [List<X>] where X implements EQ { ... }
// new implementation
implementation<X> EQ [LinkedList<X>] where X extends Number { ... }
```

Now consider the call list1.eq(list2), where both list1 and list2 have (dynamic) type LinkedList<Expr>. The implementation for List<X> may be used to resolve this call but the one for LinkedList<X> may not because the constraint Expr **extends** Number does not hold. However, JavaGI's run-time system is unable to detect this mismatch because it cannot perform constraint entailment at run time (in particular, the type argument Expr is not available because of type erasure [Bracha et al. 1998]).

Thus, JavaGI rejects the program statically because LinkedList<X> is a subtype of List<X> but the constraint X **implements** EQ of the List<X> implementation does not imply the constraint X **extends** Number of the LinkedList<X> implementation.

3.4.5 *Criterion: 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, Section 2.2 implements the EQ interface retroactively, so it is not possible to use EQ as an implementing type of any non-abstract implementation.

Disallowing implementation chains ensures decidability of constraint entailment and subtyping [Wehr and Thiemann 2008; 2009b; Wehr 2010]. Moreover, it enables efficient run-time lookup of retroactively implemented methods.

3.4.6 *Criterion: Completeness.* The implementation of an interface method must be complete, even if there exist retroactive implementations with abstract definitions for the method. That is, if a retroactive implementation of interface I contains an abstract definition of method m with $\mathsf{T}_1, \ldots, \mathsf{T}_n$ being the dispatch-relevant argument types (i.e., the receiver type and those argument types declared as implementing types in I), then the following must hold: for each sequence of non-abstract types $\mathsf{U}_1, \ldots, \mathsf{U}_n$ with $\mathsf{U}_i \leq \mathsf{T}_i$ for all $i \in \{1, \ldots, n\}$, there exists a retroactive implementation of I containing a non-abstract definition of m with $\mathsf{V}_1, \ldots, \mathsf{V}_n$ being the dispatch-relevant argument types such that $\mathsf{U}_i \leq \mathsf{V}_i$ and $\mathsf{V}_i \leq \mathsf{T}_i$ for all $i \in \{1, \ldots, n\}$. The intuition for this criterion is that the U_i are potential run-time types of the $\overline{T}$ The completeness criterion ensures that dynamic method lookup never encounters an abstract definition of some interface method.

3.4.7 *Checking the Criteria.* The JavaGI compiler checks the well-formedness criteria just described on all accessible types and implementations. At run time, however, a different set of types and implementations may be available because of subsequent edits or dynamic loading. Hence, JavaGI's run-time system re-checks the well-formedness criteria every time it loads a new type or a new set of implementations. Nevertheless, the compiler can guarantee one important property: if a program meets the well-formedness criteria at compile time and the same set of types and implementations is available at run time, then the run-time checks never fail.

3.5 Dynamic Method Lookup

At program start, JavaGI's run-time system loads all accessible implementations, checks the well-formedness criteria just explained, and installs the implementations loaded as the current pool of implementations. A dynamically loaded implementation extends this pool after checking that the well-formedness criteria still hold.

For Java call-sites (see Section 3.3), dynamic method lookup is the same as for plain Java. For JavaGI call-sites, which the compiler also marks with the interface defining the method and the argument positions of the implementing types, dynamic method lookup searches the pool of implementations for one that matches

(1) the interface in which the method is defined,
(2) the dynamic receiver type, and
(3) the dynamic types of those arguments declared as implementing types in the interface method signature.

Because of type erasure [Bracha et al. 1998], the run-time system restricts matching to the outermost type constructor and ignores type arguments. Static typing and

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{\mathit{def}}\ e \\
\mathit{def} & ::= & \mathit{cdef} \mid \mathit{idef} \mid \mathit{impl} \\
\mathit{cdef} & ::= & \texttt{class}\ C\langle\overline{X}\rangle\ \texttt{extends}\ N\ \texttt{where}\ \overline{P}\,\{\,\overline{T\,f}\ \overline{m : \mathit{mdef}}\,\} \\
\mathit{idef} & ::= & \texttt{interface}\ I\langle\overline{X}\rangle\ [\overline{Y}\ \texttt{where}\ \overline{R}]\ \texttt{where}\ \overline{P}\,\{\,\overline{m : \texttt{static}\ \mathit{msig}}\ \overline{\mathit{rcsig}}\,\} \\
\mathit{impl} & ::= & \texttt{implementation}\langle\overline{X}\rangle\ K\ [\,\overline{N}\,]\ \texttt{where}\ \overline{P}\ \{\,\texttt{static}\ \mathit{mdef}\ \ \overline{\mathit{rcdef}}\,\} \\
\mathit{rcsig} & ::= & \texttt{receiver}\,\{\overline{m : \mathit{msig}}\} \\
\mathit{rcdef} & ::= & \texttt{receiver}\,\{\overline{\mathit{mdef}}\} \\
\mathit{msig} & ::= & \langle\overline{X}\rangle\,\overline{T\,x} \to T\ \texttt{where}\ \overline{P} \\
\mathit{mdef} & ::= & \mathit{msig}\,\{e\}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
M, N & ::= & C\langle\overline{T}\rangle \mid \texttt{Object} & R, S & ::= & \overline{G}\ \texttt{implements}\ K \\
G, H & ::= & X \mid N & \mathcal{R}, \mathcal{S} & ::= & \overline{T}\ \texttt{implements}\ K \\
K, L & ::= & I\langle\overline{T}\rangle & P, Q & ::= & R \mid X\ \texttt{extends}\ T \\
T, U, V, W & ::= & G \mid K & \mathcal{P}, \mathcal{Q} & ::= & \mathcal{R} \mid T\ \texttt{extends}\ T
\end{array}
$$

$$
e, d ::= x \mid e.f \mid e.m\langle\overline{T}\rangle(\overline{e}) \mid K[\overline{T}].m\langle\overline{T}\rangle(\overline{e}) \mid \texttt{new}\ N(\overline{e}) \mid (T)\,e
$$

$$
\begin{array}{ccc}
X, Y, Z \in \mathit{TyvarName} & C, D \in \mathit{ClassName} & I, J \in \mathit{IfaceName} \\
m \in \mathit{MethodName} & f, g \in \mathit{FieldName} & x, y, z \in \mathit{VarName}
\end{array}
$$

Figure 3: Syntax of CoreGI.

the well-formedness criteria guarantee that the search just described always returns a unique most specific implementation.

The static distinction between Java call-sites and JavaGI call-sites requires that methods in retroactive implementations do not override methods defined in classes. However, the conservativeness principle postulated in Section 2.9 prevents such retroactive method overrides anyway: admitting them means that the behavior of an existing Java program could be modified by adding an appropriate implementation that overrides an internal method of some class.

4. FORMALIZATION

This section formalizes the calculus CoreGI, which captures the essential ingredients of the full JavaGI language. The formalization of the calculus specifies CoreGI's syntax, its constraint entailment and subtyping relations, its dynamic semantics, and its static semantics. CoreGI enjoys type soundness and a deterministic evaluation relation. Moreover, there exist algorithms for deciding constraint entailment, subtyping, expression typing, and program typing. The definition of CoreGI is based on that of Featherweight Generic Java (FGJ [Igarashi et al. 2001]).

4.1 Syntax

Figure 3 defines the abstract syntax of CoreGI. The various kinds of identifiers are drawn from pairwise disjoint and countably infinite sets of type variables (ranged over by X, Y, Z), class names (ranged over by C, D), interface names (ranged over by I, J), method names (ranged over by m), field names (ranged over by f, g), and expression variables (ranged over by x, y, z). Overbar notation $\overline{\xi}^n$ (or $\overline{\xi}$ for short) denotes the sequence $\xi_1 \ldots \xi_n$ for some syntactic construct ξ, where in some places commas separate the sequence items. The symbol $\bullet$ denotes the empty sequence, where some places omit the $\bullet$ altogether. Using index variables i, j, k to subscript items from a sequence implicitly assumes that the index variable ranges over the length of the sequence. Furthermore, if the same index variable subscripts items

from different sequences, then all sequences involved are implicitly assumed to be of the same length. An index variable under an overbar marks the parts that vary from sequence item to sequence item; for example, $\overline{\xi' \xi_i}$ abbreviates $\xi' \xi_1 \ldots \xi' \xi_n$. In some places, the sequence $\overline{\xi}$ stands for the set $\{\xi_1, \ldots, \xi_n\}$.

A CoreGI program *prog* is of a sequence of definitions *def* followed by a "main" expression e. A definition is either a class, interface, or implementation definition.

The type parameters $\overline{X}$ of classes, interfaces, implementations, and methods do not carry explicit bounds; instead, CoreGI exclusively uses constraint clauses of the form "`where` $\overline{P}$". For readability, code fragments omit empty lists of type parameters and type arguments "$\langle\bullet\rangle$" as well as empty constraint clauses "`where` $\bullet$".

Each class C has an explicit superclass N, where N is a class type (either an instantiated class or `Object`). If the superclass is `Object`, we sometimes omit the `extends` clause completely. The predefined class `Object` does not have a superclass and it does not define any fields or methods. The body of an ordinary class contains a sequence of field definitions $T\,f$, where T is a type and f the name of the field, followed by a sequence of method definitions $m : \mathit{mdef}$, where m is the method name and *mdef* specifies the signature *msig* and the body expression e of the method. The signature of a method consists of type parameters $\overline{X}$, value parameters $\overline{x}$ together with their types $\overline{T}$, a result type T, and constraints $\overline{P}$.

An interface I is not only parameterized over regular type parameters $\overline{X}$ but also over type parameters $\overline{Y}$, standing for the interface's implementing types. The implementation constraints $\overline{R}$ (explained shortly) attached to the implementing type parameters specify the superinterfaces of I. These *superinterface constraints* naturally generalize Java's `extends` clause for interfaces, which are not sufficiently expressive in the presence of multi-headed interfaces.

The body of an interface contains method signatures $m : \mathit{msig}$ for static methods and receiver signature *rcsig* holding the signatures of non-static methods. Unlike in full JavaGI, `receiver` declarations are matched by position, not by name; that is, the ith `receiver` declaration implicitly corresponds to the ith implementing type. For example, JavaGI allows the two `receiver` declarations in the ObserverPattern interface from Section 2.7 to be swapped like this

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

wheres CoreGI requires a fixed ordering for them:

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

Also different than in JavaGI, the CoreGI calculus does not support interface methods to be implemented directly in classes. Further, the following three syntactic conventions apply to CoreGI (but not to JavaGI) programs:

Convention 4.1 *Disjoint identifier sets for class and interface methods.* The identifier sets for class and interface methods are disjoint. Sometimes, m^{c} or m^{i} explic-

itly denotes the name of a class or interface method, respectively.

Convention 4.2 *Globally unique names of interface methods.* The names of interface methods are globally unique; that is, if some interface defines a method m then no other interface defines a method with the same name m.

Convention 4.3. Syntactic constructs that differ only in the names of bound type and expression variables are interchangeable in all contexts [Pierce 2002].

An implementation definition specifies an implementation of interface K for implementing types $\overline{N}$, where $\overline{N}$ is a sequence of class types.[15] The body of an implementation contains static methods and receiver definitions. Static methods are anonymous because they are matched by position against the static methods of the interface being implemented; that is, the ith static method in an implementation definitions provides the implementation of the ith static method in the corresponding interface.

Similar to interfaces, receiver definitions are matched by position, so the ith receiver definition corresponds to the ith implementing type. Moreover, methods inside receiver definitions are anonymous because they are matched by position against the methods in the corresponding receiver signature of the interface being implemented. For example, in an implementation of interface I, the jth method of the ith receiver definition corresponds to the jth method of the ith receiver signature of I.

Metavariables M, N range over class types, whereas G, H denote either a type variable or a class type N. Metavariables K, L range over interface types. Full types (denoted by T, U, V, W) are either G-types or interface types.

Constraints come in four forms:

(1) R, S denote *implementation constraints* that constrain only G-types. (A G-type is a type formed by the syntax rules for G in Figure 3.)

(2) P, Q denote either *subtype constraints* on type variables or R-constraints. (An R-constraint is a constraint formed by the syntax rules for R in Figure 3.)

(3) $\mathcal{R}, \mathcal{S}$ denote unrestricted implementation constraints that may constrain arbitrary types.

(4) $\mathcal{P}, \mathcal{Q}$ denote unrestricted P-constraints. (A P-constraint is a constraint formed by the syntax rules for P in Figure 3.)

With single-headed interfaces, R-constraints on class types (i.e., constraints of the form N `implements` K) are merely obfuscated syntax for trivial constraints that are unconditionally true or false. With multi-headed interfaces, however, they permit the specification of dependencies between class types and type variables. The constraint forms $\mathcal{R}$ and $\mathcal{P}$ do not occur in source programs but only as the result of applying a type substitution to some R- or P-constraint. Nevertheless, the syntactic distinction between R, P and $\mathcal{R}, \mathcal{P}$ is necessary because the proofs of the theorems to be presented in Sections 4.5–4.8. rely on it.

[15] Full JavaGI also permits single-headed interfaces to be implemented by an interface type, cf. Section 5.1.6.

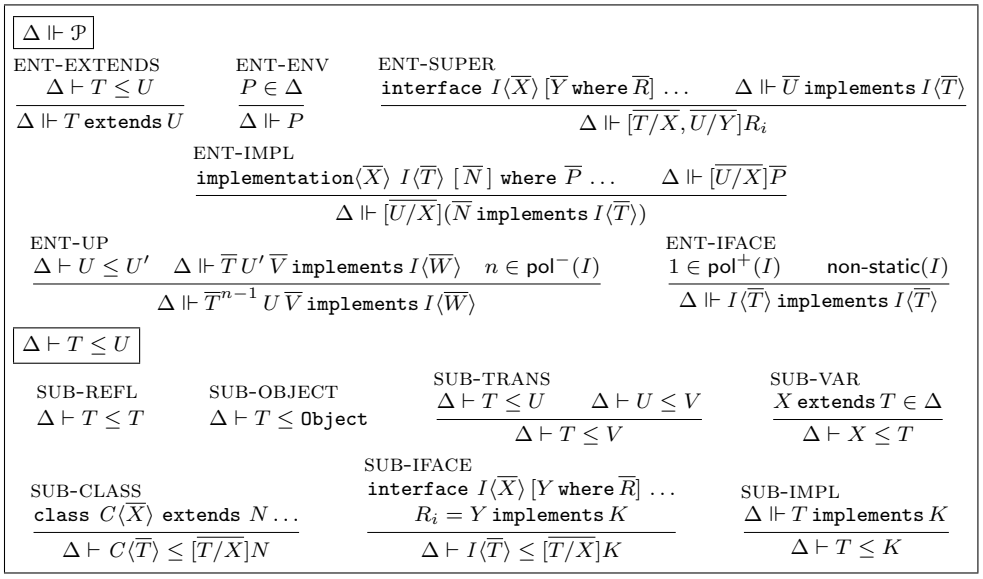

Figure 4: Constraint entailment and subtyping.

Expressions d, e include variables, field accesses, method calls, object allocations, and casts. A method call of the form $e.m\langle\overline{T}\rangle(\overline{e})$ invokes method m on receiver e with type arguments $\overline{T}$ and expression arguments $\overline{e}$.[16] Calling a static interface method takes the form $K[\overline{T}].m\langle\overline{U}\rangle(\overline{e})$, where K is the interface defining method m, $\overline{T}$ are the relevant implementing types, and $\overline{U}$ and $\overline{e}$ are the type and expression arguments, respectively.

4.2 Constraint Entailment and Subtyping

Constraint entailment (entailment for short) and subtyping play important roles in the semantics of CoreGI. Whereas constraint entailment is a purely static concept, subtyping is involved in both, the static and dynamic semantics: In the dynamic semantics, method dispatch and the evaluation of cast operations rely on a restricted form of subtyping; in the static semantics, expression typing and many other definitions depend on subtyping and on constraint entailment. This section presents a declarative, mutually recursive specification of constraint entailment and subtyping. Section 4.6 presents the corresponding algorithms.

Definition 4.4 *Type environment.* A *type environment* Δ is a finite set of type variables X and constraints P. The domain of a type environment Δ, written $\mathsf{dom}(\Delta)$, is the set of type variables contained in Δ. The notation Δ, P abbreviates $\Delta \cup \{P\}$ and Δ, X stands for $\Delta \cup \{X\}$, assuming $X \notin \mathsf{dom}(\Delta)$.

Constraint entailment, written $\Delta \Vdash \mathcal{P}$, asserts that constraint $\mathcal{P}$ holds under type environment Δ. The notation $\Delta \Vdash \overline{\mathcal{P}}$ abbreviates $(\forall i)\ \Delta \Vdash \mathcal{P}_i$. The definition of constraint entailment is mutually recursive with respect to the definition of the subtyping relation $\Delta \vdash T \leq U$, which holds if, and only if, T is a subtype of U under

[16] Full JavaGI supports inference of type arguments much as Java does, cf. Section 5.1.5.

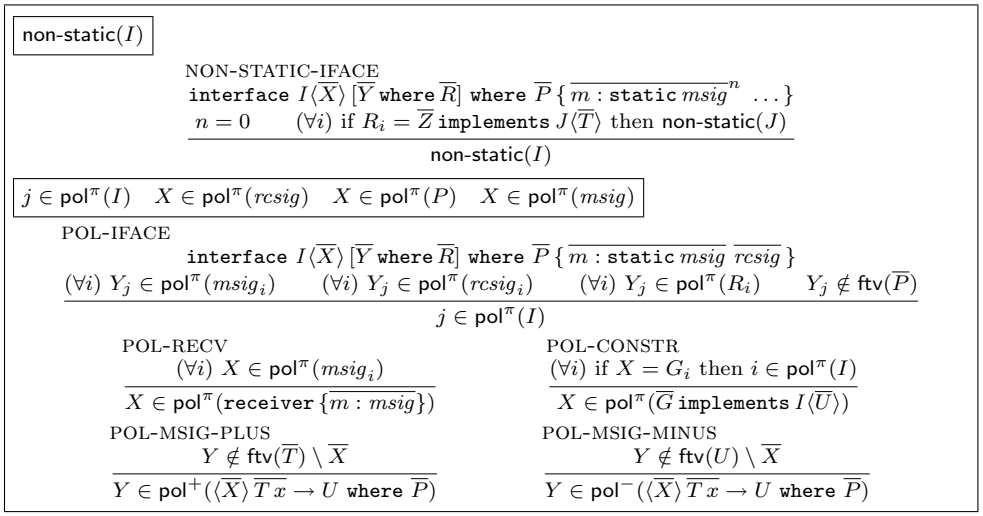

Figure 5: Restrictions on interfaces and implementing types.

type environment Δ. In some places, $\Delta \vdash \overline{T} \leq \overline{U}$ abbreviates $(\forall i)\ \Delta \vdash T_i \leq U_i$. Figure 4 defines both constraint entailment and subtyping.

The subtyping relation is reflexive and transitive, and it installs `Object` as a supertype of every other type. A type variable X is a subtype of T if the type environment contains the constraint X `extends` T. Moreover, a class type is a subtype of its direct superclass. The premise `class` $C\langle\overline{X}\rangle$ `extends` $N \ldots$ of rule SUB-CLASS is an abbreviation for a predicate ensuring that there exists a corresponding class definition in the program under consideration. We use similar abbreviations throughout the rest of this article.

Rule SUB-IFACE formulates subtyping on interface types in terms of superinterface constraints. The rule is only applicable to single-headed interfaces because only these interfaces may serve as types (see Section Section 2.7 and Section 4.4) . Finally, rule SUB-IMPL integrates constraint entailment into the subtyping relation by deriving $\Delta \vdash T \leq K$ from $\Delta \Vdash T$ `implements` K.

Constraint entailment in Figure 4 solves subtype constraints by invoking the subtyping relation (rule ENT-EXTENDS), whereas rule ENT-ENV specifies that a constraint from the type environment is always considered valid. Rule ENT-SUPER states that a constraint implies all superinterface constraints of its corresponding interface. The notation $[\overline{T/X}]$ denotes the capture-avoiding *type substitution* that replaces type variables X_i with types T_i. Metavariables φ and ψ range over type substitutions.

Rule ENT-IMPL defines how an implementation definition establishes validity of a constraint. Rule ENT-UP promotes a type on the left-hand side of an implementation constraint to a supertype. To ensure type soundness, the rule requires that the corresponding implementing type does not occur in covariant positions of the interface (premise $n \in \mathsf{pol}^-(I)$). This rule makes again use of the subtyping relation defined in the same figure. Rule ENT-IFACE is a kind of reflexivity rule. However, the rule only fires for interfaces without binary methods (premise $1 \in \mathsf{pol}^+(I)$) and without static methods (premise $\mathsf{non\text{-}static}(I)$) to ensure type soundness.

Figure 5 specifies the two auxiliary predicates non-static and pol. The non-static predicate asserts that neither the given interface I nor any of its superinterfaces contains a static method. The pol predicate defines the *polarity* of the ith implementing type of interface I as positive (or negative), written $i \in \mathsf{pol}^+(I)$ (or $i \in \mathsf{pol}^-(I)$), if the implementing type does not occur in contravariant (or covariant) positions. We let π range over $+$ and $-$. The notation $\mathsf{ftv}(\xi)$ denotes the set of type variables free in some syntactic construct ξ. The definition of $j \in \mathsf{pol}^\pi(I)$ by rule POL-IFACE in Figure 5 relies on the polarity of an implementing type variable X in receiver signatures ($X \in \mathsf{pol}^\pi(\mathit{rcsig})$), constraints ($X \in \mathsf{pol}^\pi(P)$), and method signatures ($X \in \mathsf{pol}^\pi(\mathit{msig})$). The definition of the latter by rules POL-MSIG-PLUS and POL-MSIG-MINUS depends on a restriction stating that an implementing type variable may appear in a method signature only at the top level of the result type and at the top level of the argument types. Section 4.4.3 formalizes this restriction as well-formedness criterion WF-IFACE-3.

4.3 Dynamic Semantics

This section presents a structural operational semantics [Plotkin 1981] defining the run-time behavior of CoreGI programs.

4.3.1 *Method Lookup.* Figure 6 formalizes dynamic method lookup. The relation $\mathsf{getmdef}^{\mathsf{c}}(m, N)$ performs dynamic lookup of class method m on a receiver with run-time type N. If possible, it returns the definition of m directly contained in N (rule DYN-MDEF-CLASS-BASE). Otherwise, it continues the search in N's superclass (rule DYN-MDEF-CLASS-SUPER). The search stops when it reaches `Object` because there is no matching rule.

For non-static interface methods, $\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N})$ performs lookup of a retroactively implemented method m on receiver type N and actual parameter types $\overline{N}$. For static interface methods, $\mathsf{getsmdef}(m, K, \overline{U})$ searches for method m in an implementation definition matching interface K and implementing types $\overline{U}$. The definitions of $\mathsf{getmdef}^{\mathsf{i}}$ and getsmdef rely on several auxiliaries from Figure 6:

— $N_1 \sqcup N_2 = M$ computes the least upper bound M of class types N_1 and N_2.

— $\bigsqcup \mathscr{N} = N$ computes the least upper bound N of a set $\mathscr{N}$ of class types. If $\mathscr{N}$ is not empty then the least upper bound is unique and always exists.

— $\mathsf{resolve}_X(\overline{T}, \overline{N}) = N^?$ resolves implementing type X with respect to formal parameter types $\overline{T}$ and run-time parameter types $\overline{N}$ as the optional class type $N^?$. The notation $\xi^?$ denotes an optional construct; that is, $\xi^?$ is either a regular ξ or the special symbol nil.

The definition of resolve constructs a set $\mathscr{C}$ containing those run-time parameter types N_i such that the ith formal parameter dispatches on X (i.e., $T_i = X$). If the set $\mathscr{C}$ is not empty (rule RESOLVE-NON-EMPTY), the resolution of X is the least upper bound $\bigsqcup \mathscr{C}$. Otherwise (rule RESOLVE-EMPTY), X does not occur in the formal parameter types $\overline{T}$, so resolve returns nil. There is a restriction ensuring that the implementing type X does not occur nested inside one of the formal parameter types T_i (see well-formedness criterion WF-IFACE-3 in Section 4.4.3).

— $\mathsf{least\text{-}impl}\mathscr{M}$ computes the least element of a set $\mathscr{M}$ containing pairs of substitutions and implementations. The pair (φ, impl) is considered smaller than the

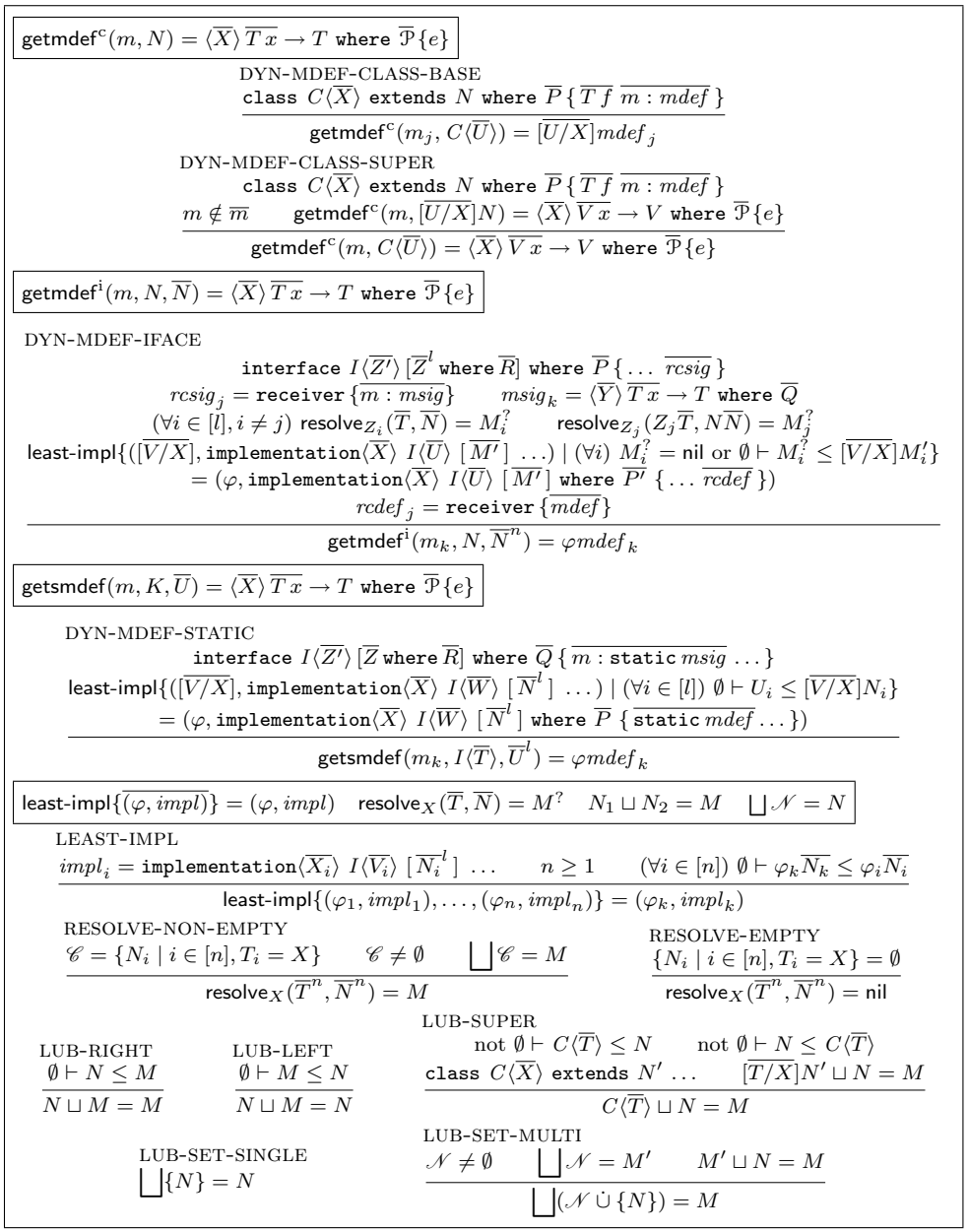

Figure 6: Dynamic method lookup.

pair $(\varphi', impl')$ if, and only if, the implementing types of $impl$ under substitution φ are pointwise subtypes of the implementing types of $impl'$ under substitution φ'. The notation $[n]$ (used by rule LEAST-IMPL) denotes the set $\{1, \ldots, n\}$, where $n \in \mathbb{N}$. For $n = 0$, $[n] = \emptyset$.
There are several restrictions ensuring that least-impl always finds a unique solution when invoked by $\mathsf{getmdef}^{\mathsf{i}}$ or getsmdef. Sections 3.4.1, 3.4.2, and 3.4.3 already discussed these restrictions informally; Section 4.4.3 defines them formally as well-formedness criteria WF-PROG-1, WF-PROG-2, and WF-PROG-3.

With these auxiliaries in place, rule DYN-MDEF-IFACE defines $\mathsf{getmdef}^{\mathsf{i}}(m, N, \overline{N})$ as follows:

—First, $\mathsf{getmdef}^{\mathsf{i}}$ retrieves the interface I and the receiver $rcsig_j$ defining the method m.
—Then, it uses resolve to compute, for each implementing type variable Z_i, an optional least upper bound $M_i^?$ of all argument types contributing to the resolution of the ith implementing type.
—Next, it collects all implementations of I whose implementing types are pointwise supertypes of the $M_i^?$s. (If $M_i^?$ is nil, then every type is considered a supertype of $M_i^?$ because the ith implementing type does not occur in m's signature.)
—Finally, $\mathsf{getmdef}^{\mathsf{i}}$ selects among all these implementations the one with least implementing types.

The definition of $\mathsf{getsmdef}(m, K, \overline{U})$ in rule DYN-MDEF-STATIC is similar to that of $\mathsf{getmdef}^{\mathsf{i}}$ but simpler: getsmdef does not need to resolve the implementing types but gets them explicitly through the types $\overline{U}$. Thus, getsmdef just uses least-impl to choose the least implementation among all implementation definitions matching K and $\overline{U}$.

4.3.2 *Evaluation.* The definition of CoreGI's dynamic semantics is now straightforward and given in Figure 7. Values (ranged over by v, w) and call-by-value evaluation contexts (denoted by $\mathcal{E}$) are defined in the obvious way. Unlike FGJ, CoreGI uses a call-by-value evaluation order to ensure deterministic evaluation. The notation $\mathcal{E}[e]$ denotes the replacement of $\mathcal{E}$'s hole $\Box$ with expression e.

The *top-level evaluation* relation $e \longmapsto e'$ reduces an expression e at the top level to e'. Rule DYN-FIELD deals with field accesses $\texttt{new}\, N(\overline{v}).f_i$. The auxiliary relation $\mathsf{fields}(N) = \overline{T\, f}$, also defined in Figure 7, returns the fields declared by the superclasses of N and N itself. CoreGI assumes that the ith constructor argument v_i corresponds to the field $T_i\, f_i$, so $\texttt{new}\, N(\overline{v}).f_i$ reduces to v_i. Rules DYN-INVOKE-CLASS, DYN-INVOKE-IFACE, and DYN-INVOKE-STATIC handle invocations of class methods, non-static interface methods, and static interface methods, respectively. The notation $[\overline{e/x}]$ denotes the capture-avoiding expression substitution that replaces expression variables x_i with expressions e_i. Among the rules DYN-INVOKE-CLASS and DYN-INVOKE-IFACE, at most one is applicable because the identifier sets of class (m^{c}) and interface (m^{i}) methods are disjoint (see Convention 4.1). Finally, rule DYN-CAST permits casts from $\texttt{new}\, N(\overline{v})$ to type T if N is a subtype of T.

The *proper evaluation* relation $e \longrightarrow e'$ reduces an expression e to e' by using a suitable evaluation context $\mathcal{E}$ together with the top-level evaluation relation $\longmapsto$.

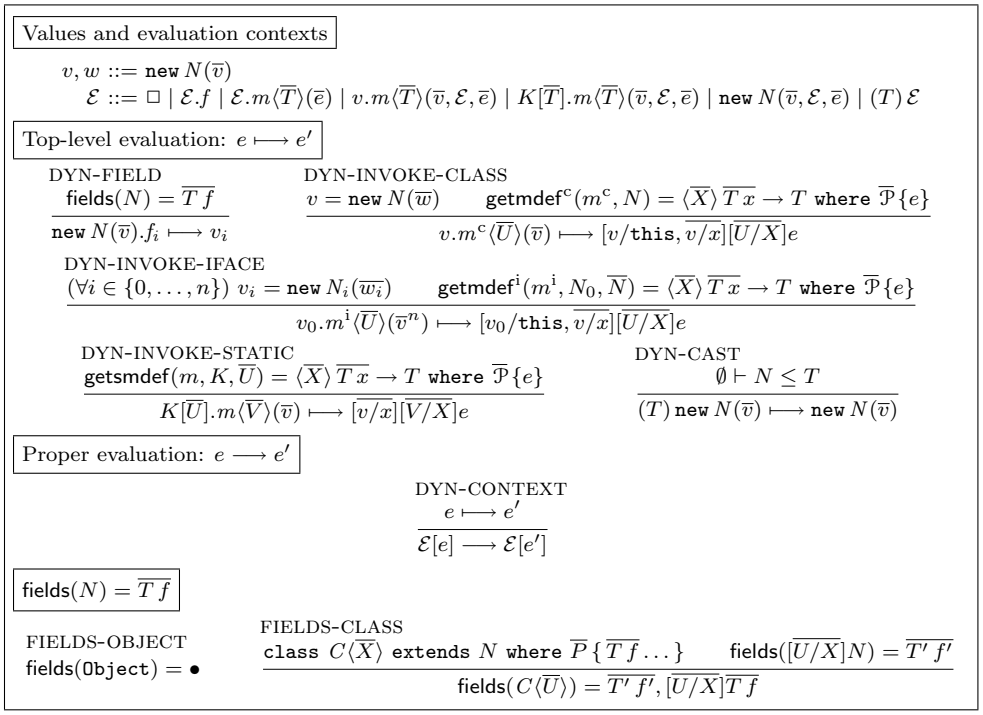

Figure 7: Dynamic semantics.

Remark. Several places in the definition of the dynamic semantics rely on CoreGI's subtyping relation. Except for the premise of rule DYN-CAST in Figure 7, all uses of the subtyping relation have the form $\emptyset \vdash T \le N$; that is, the type environment is empty and only class types appear as possible supertypes. In these cases, the full subtyping relation is not needed; instead, plain inheritance between classes and an additional rule covering the case $N = \texttt{Object}$ suffices.[17]

4.4 Static Semantics

This section presents a declarative specification of CoreGI's type system. We defer the definition of a typechecking algorithm until Section 4.7.

All types and constraints occurring in a type-correct CoreGI program must be well-formed. Formally, a type T or constraint $\mathcal{P}$ is well-formed under type environment Δ if, and only if, $\Delta \vdash T\ \mathsf{ok}$ or $\Delta \vdash \mathcal{P}\ \mathsf{ok}$, respectively, holds (see Figure 8). Often $\Delta \vdash \overline{T}\ \mathsf{ok}$ and $\Delta \vdash \overline{\mathcal{P}}\ \mathsf{ok}$ abbreviate $(\forall i)\ \Delta \vdash T_i\ \mathsf{ok}$ and $(\forall i)\ \Delta \vdash \mathcal{P}_i\ \mathsf{ok}$, respectively. Rule OK-IFACE ensures that only single-headed interfaces form interface types (see Section 2.7). Well-formedness of a constraint $\overline{T}\ \texttt{implements}\ I\langle\overline{U}\rangle$ (rule OK-IMPL-CONSTR) not only demands that $\overline{T}, \overline{U}$ are well-formed but also that the constraints of the interface I are fulfilled.

The relation $\mathsf{mtype}_\Delta(m, T)$, defined in Figure 9, looks up the signature of method m for receiver type T. Rule MTYPE-CLASS handles class methods m^{c}. Unlike the corresponding rule for FGJ, lookup of class methods does not ascend the inheri-

[17] The definition of inheritance between classes is standard (see Figure 14).

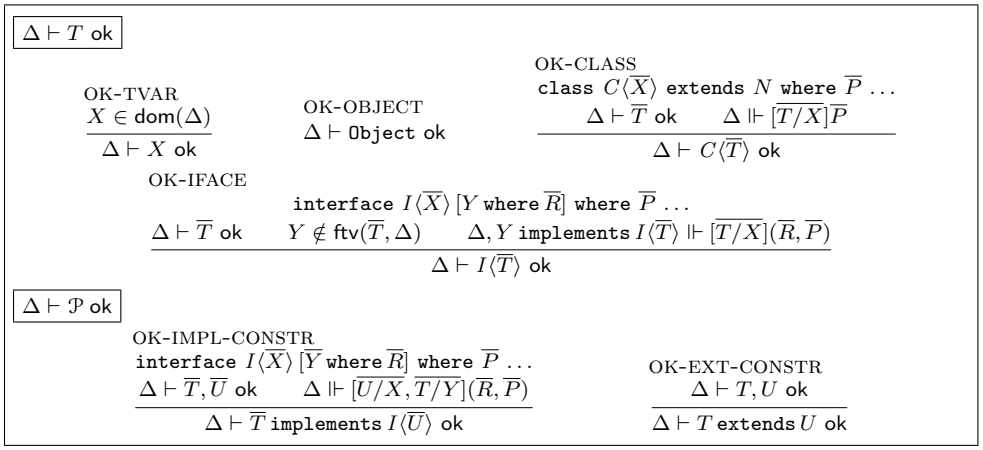

Figure 8: Well-formedness of types and constraints.

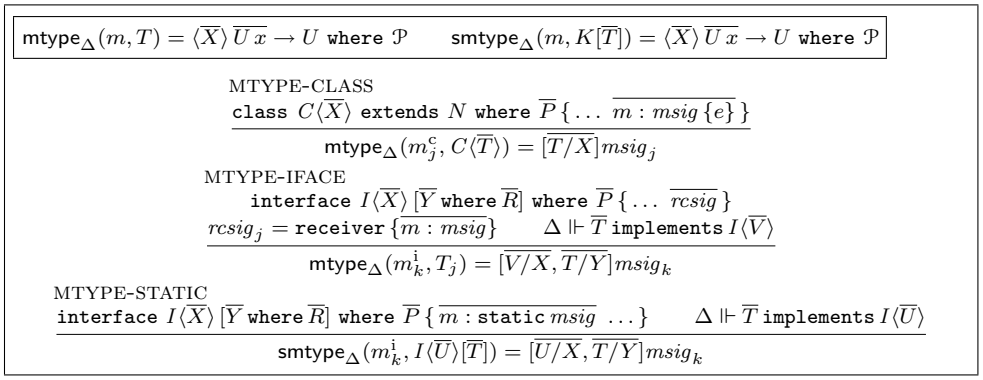

Figure 9: Method typing.

tance hierarchy of classes because CoreGI's typing rules (explained shortly) permit subsumption on the receiver. Rule MTYPE-IFACE handles interface methods m^{i} by searching the interface and the receiver defining the method and asserting validity of the corresponding implementation constraint, possibly "guessing" the types $\overline{V}$ and some of the types $\overline{T}$. Figure 9 also defines the relation $\mathsf{smtype}_\Delta(m, I\langle\overline{U}\rangle[\overline{T}])$, which looks up the signature of static method m defined in interface I under type parameters $\overline{U}$ and implementing types $\overline{T}$.

4.4.1 *Expression Typing.* Expression typing, written $\Delta; \Gamma \vdash e : T$, states that under type environment Δ and variable environment Γ, expression e has type T. Variable environments are defined as follows:

Definition 4.5 *Variable environment.* A *variable environment* Γ is a finite mapping from variables x to types T. The notation $\Gamma, x : T$ extends Γ with a mapping from x to T, assuming x is not already bound in Γ. The notation $\Gamma(x)$ denotes the type T such that Γ maps x to T. It assumes that Γ contains a binding for x.

Figure 10 defines the expression typing judgment. Typechecking a field access $e.f_j$ looks up the type of field f_j in the fields declared by C (rule EXP-FIELD). There is no

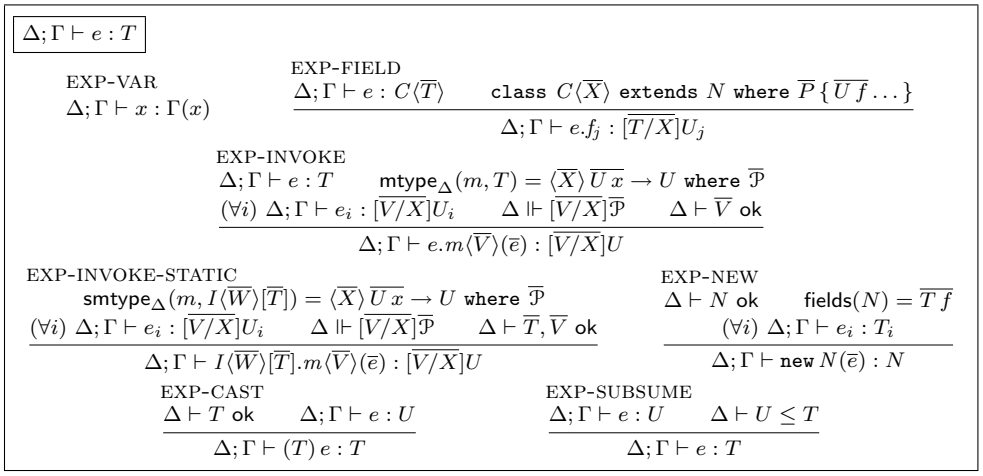

Figure 10: Expression typing.

need to search the superclasses of C for a definition of f_j because rule EXP-SUBSUME permits lifting the type of e to some supertype. Thanks to mtype and smtype from Figure 9, typechecking method invocations is straightforward (rules EXP-INVOKE and EXP-INVOKE-STATIC).

Rule EXP-NEW handles an object allocation $\texttt{new}\,N(\overline{e})$ by asserting that N is well-formed and by checking that the ith argument e_i is type correct with respect to the type of the ith field declaration returned by $\mathsf{fields}(N)$. Unlike FGJ, which has three rules for cast expressions to differ between upcasts, downcasts, and stupid casts, CoreGI uses a single rule for casts because they are not in the focus of the formalization.

4.4.2 *Program Typing.* Figure 11 specifies the well-formedness rules for programs, including several auxiliary relations.

—The relation $\Delta \vdash msig \le msig'$ extends subtyping to method signatures by treating return types covariantly and argument types invariantly (see rule SUB-MSIG).

—The relation $\mathsf{override\text{-}ok}_\Delta(m : msig, N)$ asserts that class type N correctly overrides method m with signature $msig$ (see rule OK-OVERRIDE).

—The relations $\Delta \vdash msig\ \mathsf{ok}$, $\Delta \vdash mdef\ \mathsf{ok}$, and $\Delta \vdash rcsig\ \mathsf{ok}$ assert well-formedness of method signatures, method definitions, and receiver signatures, respectively (see rules OK-MSIG, OK-MDEF, and OK-RCSIG, respectively).

—The relation $\Delta \vdash m : mdef\ \mathsf{ok\,in}\ N$ asserts that the definition $mdef$ of method m in class N is well-formed (see rule OK-MDEF-IN-CLASS).

—The relation $\Delta \vdash mdef\ \mathsf{implements}\ msig$ asserts that method definition $mdef$ is a valid implementation of signature $msig$ (see rule IMPL-METH).

—The relation $\Delta \vdash rcdef\ \mathsf{implements}\ rcsig$ asserts that receiver definition $rcdef$ properly implements all methods from receiver signature $rcsig$ (see rule IMPL-RECV). As already discussed in Section 4.1, methods in receiver definitions are matched by position against methods in receiver signatures.

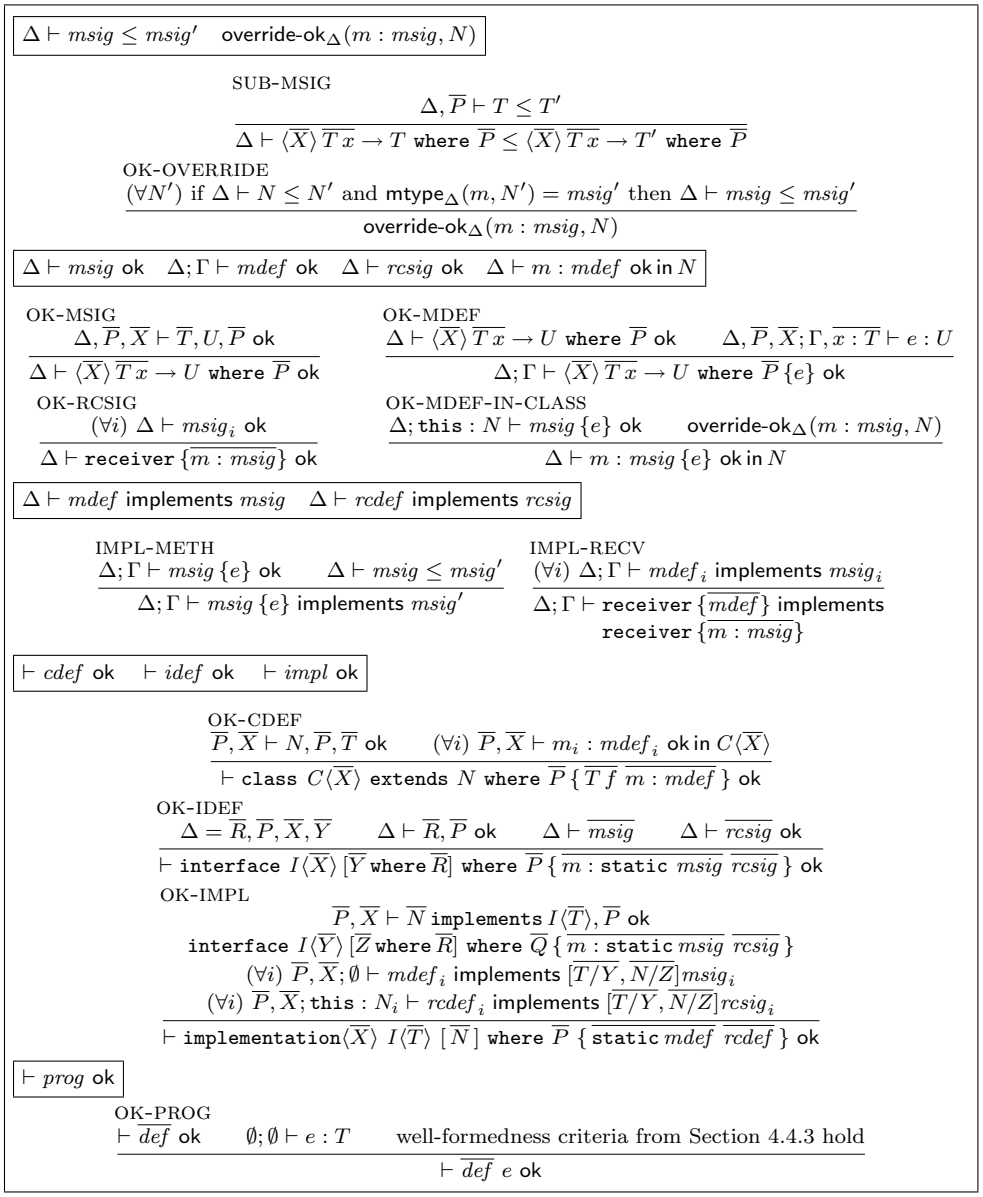

Figure 11: Well-formedness of programs.

—The relations $\vdash$ *cdef* ok, $\vdash$ *idef* ok, and $\vdash$ *impl* ok assert well-formedness of classes, interfaces, and implementations, respectively (see rules OK-CDEF, OK-IDEF, and OK-IMPL, respectively).

—The relation $\vdash$ *prog* ok asserts well-formedness of programs (see rule OK-PROG). Well-formedness of programs requires several additional well-formedness criteria, as specified in the next section. (For the full JavaGI language, Section 3.4 already discussed some of them informally.)

4.4.3 *Additional Well-formedness Criteria.* The additional well-formedness criteria for CoreGI are divided into criteria that apply to classes, interfaces, implementations, whole programs, and type environments.

Criteria for Classes. For each

$$\texttt{class}\ C\langle\overline{X}\rangle\ \texttt{extends}\ N\ \texttt{where}\ \overline{P}\,\{\,\overline{T\,f}^{\,n}\ \overline{m : mdef}^{\,l}\,\}$$

the following well-formedness criteria must hold:

WF-CLASS-1 The field names, including names of inherited fields, are pairwise disjoint. That is, $i \neq j \in [n]$ implies $f_i \neq f_j$ and $\mathsf{fields}(N) = \overline{U\,g}$ implies $\overline{f} \cap \overline{g} = \emptyset$.

WF-CLASS-2 The method names $\overline{m}$ are pairwise disjoint. That is, $i \neq j \in [l]$ implies $m_i \neq m_j$.

Criterion WF-CLASS-1 states that CoreGI does not support field shadowing, whereas WF-CLASS-2 rules out method overloading (together with rule OK-OVERRIDE from Figure 11). Both restrictions are not present in the full JavaGI language.

Criteria for Interfaces. The predicate $\mathsf{at\text{-}top}(\overline{X}, T)$ ensures that type T is either one of the type variables in $\overline{X}$ or none of these type variables occur in T at all.

Definition 4.6. $\mathsf{at\text{-}top}(\overline{X}, T)$ holds if, and only if, $\overline{X} \cap \mathsf{ftv}(T) = \emptyset$ or $T \in \overline{X}$.

The well-formedness criteria for interfaces then require that for each

$$\texttt{interface}\ I\langle\overline{X}\rangle\ [\overline{Y}\ \texttt{where}\ \overline{R}]\ \texttt{where}\ \overline{P}\,\{\,\overline{m : \texttt{static}\ msig}\ \ \overline{rcsig}\,\}$$

the following conditions must hold:

WF-IFACE-1 The names $\overline{m}$ of static methods are pairwise disjoint.

WF-IFACE-2 In all superinterface constraints $\overline{G}\ \texttt{implements}\ K \in \overline{R}$, the implementing types $\overline{Y}$ do not occur in K and the types $\overline{G}$ are pairwise distinct type variables from $\overline{Y}$; that is, $\overline{Y} \cap \mathsf{ftv}(K) = \emptyset$ and $\overline{G} \subseteq \overline{Y}$ and $G_i \neq G_j$ for $i \neq j$.

WF-IFACE-3 In all method signatures $\langle\overline{Z}\rangle\,\overline{T\,x} \to U\ \texttt{where}\ \overline{Q}$ contained in $\overline{rcsig}$, the implementing types $\overline{Y}$ may occur only at the top level of $\overline{T}$ and U, and they do not appear in $\overline{Q}$; that is, $\mathsf{at\text{-}top}(\overline{Y}, T_i)$ for all i and $\mathsf{at\text{-}top}(\overline{Y}, U)$ and $\mathsf{ftv}(\overline{Q}) \cap \overline{Y} = \emptyset$.

Criterion WF-IFACE-1 prevents overloading of static interface methods. (It is not necessary to include inherited method in this check because invocations of static interface methods are always qualified with the interface defining the method.) The full JavaGI language does not have this restriction. Criterion WF-IFACE-2 restricts the form of superinterface constraints to simplify the superinterface relation.

The last criterion WF-IFACE-3 limits implementing types in method signatures to appear only at the top level of the result and argument types. If implementing types could occur nested inside argument types, then it would be impossible to implement method dispatch under Java's type erasure semantics [Bracha et al. 1998]. Nested occurrences of implementing types in result positions would cause loss of minimal types because result types may vary covariantly but generic types are invariant with respect to their argument types [Wehr 2010, page 46]. Last but not least,

```
class C extends Object {}                    interface J [X where X implements I] {}
interface I [X] {                            implementation J [C] {}
  receiver { m : • → Object }                new C().m()
}
```

Figure 12: Illegal CoreGI program missing an implementation of interface I for class C.

implementing types in constraints of method signatures would cause unsoundness [Wehr 2010, pages 46–47].

Criteria for Implementations. The specification of the well-formedness criteria for implementation definitions requires the introduction of an alternative formulation of constraint entailment and subtyping. This alternative formulation is called *quasi algorithmic* because it constitutes a first step towards an algorithm for checking constraint entailment and subtyping.

Quasi-algorithmic constraint entailment is needed to ensure that an implementation of some interface I comes with appropriate implementations for all superinterfaces of I. As an example, consider the program in Figure 12. It fails at run time because there is no implementation of interface I for class C that could provide the code for m, so the expression `new` $C().m()$ is stuck. However, the typing rules for expressions (Figure 10) accept the expression `new` $C().m()$ because the constraint C `implements` I holds by rules ENT-SUPER and ENT-IMPL from Figure 4. The root of the problem is that there exists an implementation of interface J for class C without a suitable implementation of J's superinterface I.

A failed attempt to deal with the problem for the program in Figure 12 is to require the following condition:

WF-IMPL-1-INFORMAL-WRONG

> "For every `implementation`$\langle\overline{X}\rangle$ J [N] `where` $\overline{P}$ ... the corresponding superinterface constraint N `implements` I must hold under type environment $\overline{P}$; that is, $\overline{P} \Vdash N$ `implements` I."

But $\overline{P} \Vdash N$ `implements` I *always* holds by rule ENT-SUPER because rules ENT-IMPL and ENT-ENV allow us to derive $\overline{P} \Vdash N$ `implements` J.

A similar problem arises with Haskell type classes when checking that suitable instance definitions for all superclasses of a given type class exist [Wehr 2005].[18] In the context of Haskell, Sulzmann [2006] suggests a restricted form of constraint entailment to check for superclass instances.

We follow Sulzmann's approach and use quasi-algorithmic constraint entailment to check for appropriate implementations of superinterfaces. It is an open question whether it is possible to use the declarative form of constraint entailment instead. Figure 13 and Figure 14 define quasi-algorithmic constraint entailment and subtyping, respectively, together with several auxiliary relations.

— Quasi-algorithmic constraint entailment, written $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$, asserts validity of constraint $\mathcal{P}$ under type environment Δ. The idea of quasi-algorithmic entailment is to restrict derivations of declarative entailment (Figure 4) such that consecutive

[18] Haskell's type classes and instance definitions are analogous to JavaGI's generalized interfaces and implementation definitions, respectively. See Section 7.1 for details.

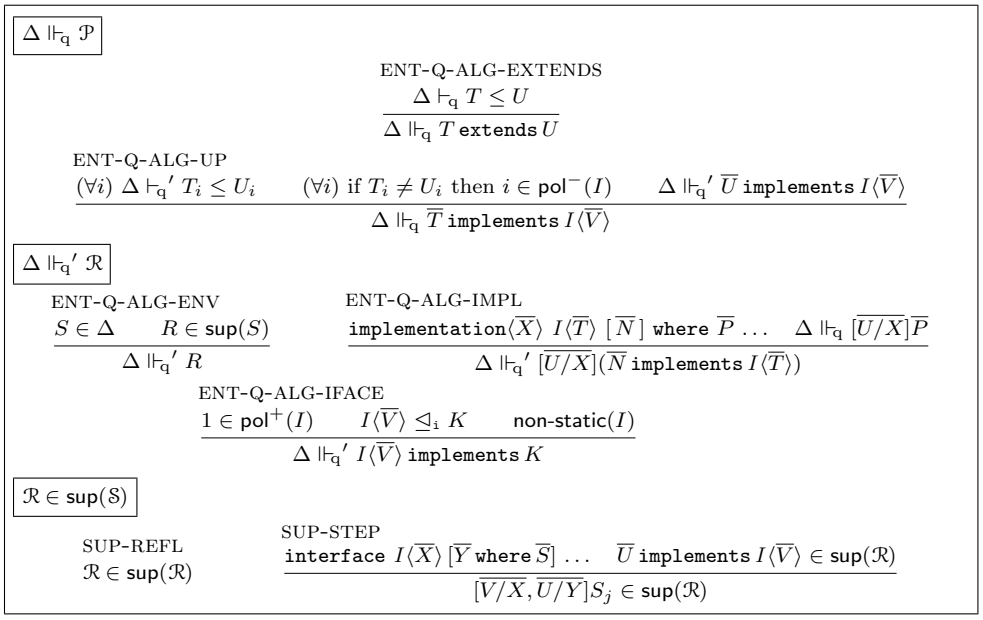

Figure 13: Quasi-algorithmic constraint entailment.

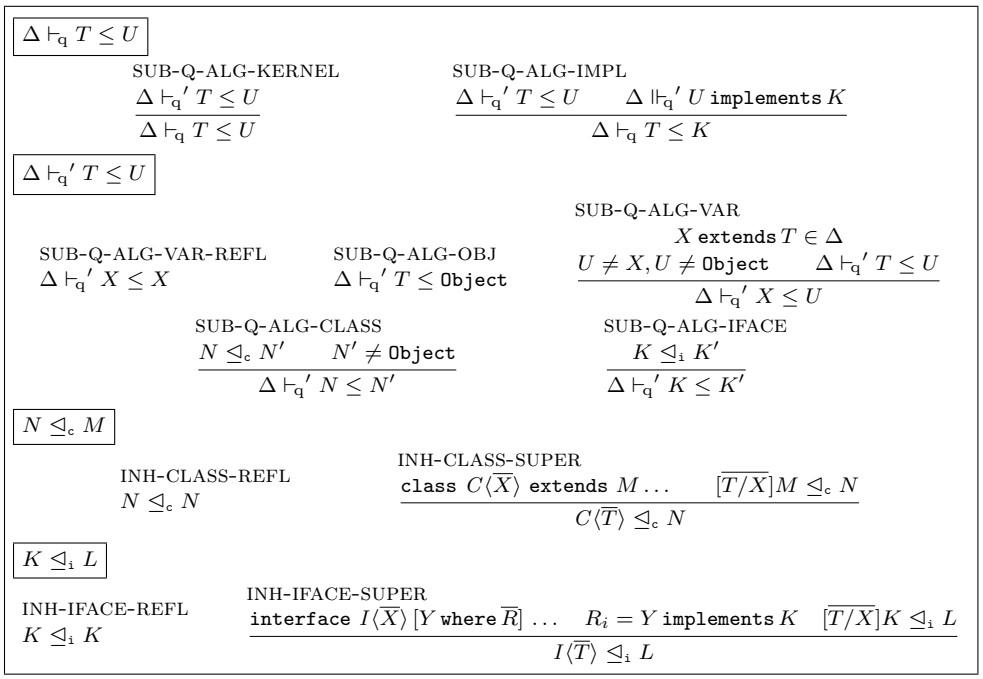

Figure 14: Quasi-algorithmic subtyping and inheritance.

applications of rule ENT-UP are merged into an application of a single rule, and that rule ENT-SUPER is applied only to constraints originally established by rule ENT-ENV or rule ENT-IFACE. In Figure 13, rule ENT-Q-ALG-UP mimics consecutive applications of rule ENT-UP: it establishes validity of a constraint $\overline{T}\,\texttt{implements}\,I\langle\overline{V}\rangle$ by first lifting all T_j pointwise to supertypes U_j, thereby respecting j's polarity in I, and then solving the resulting constraint $\overline{U}\,\texttt{implements}\,I\langle\overline{V}\rangle$.

— The *kernel of quasi-algorithmic entailment*, written $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{P}$, is a subset of the quasi-algorithmic entailment relation. Rule ENT-Q-ALG-ENV simulates an application of rule ENT-ENV followed by zero or more applications of rule ENT-SUPER, whereas rule ENT-Q-ALG-IFACE imitates an application of rule ENT-IFACE followed by zero or more applications of rule ENT-SUPER.

— The auxiliary relation $\mathcal{R} \in \texttt{sup}(\mathcal{S})$ states that $\mathcal{R}$ is a *super constraint* of $\mathcal{S}$. Super constraints arise either through reflexivity (rule SUP-REFL) or through superinterface constraints (rule SUP-STEP).

— Quasi-algorithmic subtyping, written $\Delta \vdash_{\mathrm{q}} T \leq U$, states that T is a subtype of U under type environment Δ. Quasi-algorithmic subtyping distinguishes two cases: Rule SUB-Q-ALG-KERNEL states that quasi-algorithmic subtyping includes its kernel variant (explained next), and rule SUB-Q-ALG-IMPL establishes a subtyping relationship between type T and interface type K by lifting T to U and then solving the constraint $U\,\texttt{implements}\,K$. Different to rule ENT-Q-ALG-UP, there is no polarity check.

— The *kernel of quasi-algorithmic subtyping*, written $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, is a subset of the quasi-algorithmic subtyping relation that does not include subtyping implied by constraint entailment. Essentially, the kernel of quasi-algorithmic subtyping corresponds to FGJ's subtyping relation extended with interface inheritance. The side conditions "$U \neq X, U \neq \texttt{Object}$ in rule SUB-Q-ALG-VAR and "$N' \neq \texttt{Object}$" in rule SUB-Q-ALG-CLASS ensure that the kernel of quasi-algorithmic subtyping is syntax-directed; that is, given a derivation $\mathcal{D}$ of $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, the two types T and U uniquely determine the last rule of $\mathcal{D}$.

— The relation $N \trianglelefteq_{\mathrm{c}} M$ denotes *class inheritance* between class types N and M, whereas $K \trianglelefteq_{\mathrm{i}} L$ denotes *interface inheritance* between interface types K and L. Rule INH-IFACE-SUPER expresses non-trivial inheritance between interface types through superinterface constraints. The rule is only applicable to single-headed interfaces because multi-headed interfaces do not form valid types. The notation $\overline{N} \trianglelefteq_{\mathrm{c}} \overline{M}$ abbreviates $(\forall i)\ N_i \trianglelefteq_{\mathrm{c}} M_i$.

With quasi-algorithmic constraint entailment, the condition to ensure that all superinterfaces are properly implemented for the program in Figure 12 now reads as follows (cf. condition WF-IMPL-1-INFORMAL-WRONG, page 35):

WF-IMPL-1-INFORMAL

> "For every $\texttt{implementation}\langle\overline{X}\rangle\ J\ [\,N\,]\ \texttt{where}\ \overline{P}\ \ldots$ the corresponding superinterface constraint $N\,\texttt{implements}\,I$ must hold under type environment $\overline{P}$ *with respect to quasi-algorithmic constraint entailment*; that is, $\overline{P} \Vdash_{\mathrm{q}} N\,\texttt{implements}\,I$."

Indeed, unlike WF-IMPL-1-INFORMAL-WRONG, this criterion detects that the program in Figure 12 misses an implementation of I for C: there exists no derivation

$\boxed{j \in \mathsf{disp}(I) \quad Y \in \mathsf{disp}(\mathit{rcsig}) \quad Y \in \mathsf{disp}(P) \quad Y \in \mathsf{disp}(\mathit{msig})}$

DISP-IFACE
$$\frac{\texttt{interface}\ I\langle\overline{X}\rangle\ [\overline{Y}^{n}\ \texttt{where}\ \overline{R}^{m}]\ \texttt{where}\ \overline{P}\,\{\ldots\ \overline{\mathit{rcsig}}^{n}\} \qquad (\forall i \in [n], i \neq j)\ Y_j \in \mathsf{disp}(\mathit{rcsig}_i) \qquad (\forall i \in [m])\ Y_j \in \mathsf{disp}(R_i)}{j \in \mathsf{disp}(I)}$$

DISP-RCSIG
$$\frac{(\forall i)\ Y \in \mathsf{disp}(\mathit{msig}_i)}{Y \in \mathsf{disp}(\texttt{receiver}\,\{\overline{\mathit{msig}}\})}$$

DISP-CONSTR
$$\frac{(\forall i)\ \text{if } G_i = Y \text{ then } i \in \mathsf{disp}(I)}{Y \in \mathsf{disp}(\overline{G}\,\texttt{implements}\,I\langle\overline{V}\rangle)}$$

DISP-MSIG
$$\frac{Y \notin \overline{X} \qquad Y \in \overline{T}}{Y \in \mathsf{disp}(\langle\overline{X}\rangle\,\overline{T}\,x \to T\ \texttt{where}\ \overline{P})}$$

Figure 15: Dispatch types and positions.

$\boxed{\Delta \vdash G_1 \sqcap G_2 = H}$

GLB-LEFT
$$\frac{\Delta \vdash G_1 \leq G_2}{\Delta \vdash G_1 \sqcap G_2 = G_1}$$

GLB-RIGHT
$$\frac{\Delta \vdash G_2 \leq G_1}{\Delta \vdash G_1 \sqcap G_2 = G_2}$$

Figure 16: Greatest lower bounds.

for $\emptyset \Vdash_{\mathsf{q}} C\ \texttt{implements}\ I$.

Before defining the well-formedness criteria for implementation definitions, Figure 15 introduces the notion of *dispatch types*. The jth implementing type of interface I is a dispatch type, written $j \in \mathsf{disp}(I)$, if it appears in every non-static method signature of I or one of its superinterfaces as the receiver or at the top level of some argument type. In other words: if m is a non-static method of I or any of its superinterfaces, then $j \in \mathsf{disp}(I)$ guarantees that every invocation of m resolves the jth implementing type of I. The auxiliary relations $Y \in \mathsf{disp}(\mathit{rcsig})$, $Y \in \mathsf{disp}(P)$, and $Y \in \mathsf{disp}(\mathit{msig})$ assert that the implementing type variable Y is a dispatch type with respect to a receiver *rcsig*, a constraint P, and a method signature *msig*, respectively.

The well-formedness criteria for implementations now require that for each

$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{V}\rangle\ [\overline{N}]\ \texttt{where}\ \overline{P}\ \ldots$$

the following conditions must hold:

WF-IMPL-1 There exist suitable implementations for all superinterfaces of I; that is, if $\mathcal{Q} \in \mathsf{sup}(\overline{N}\,\texttt{implements}\,I\langle\overline{V}\rangle)$ then $\overline{P} \Vdash_{\mathsf{q}} \mathcal{Q}$.

WF-IMPL-2 The dispatch types among $\overline{N}$ fully determine the type variables $\overline{X}$; that is $\overline{X} \subseteq \mathsf{ftv}(\{N_i \mid i \in \mathsf{disp}(I)\})$.

WF-IMPL-3 In all constraints $\overline{G}\,\texttt{implements}\,K \in \overline{P}$, the types $\overline{G}$ are type variables from $\overline{X}$; that is, $\overline{G} \subseteq \overline{X}$.

As already discussed, criterion WF-IMPL-1 ensures that suitable implementations for all relevant superinterfaces exist. The two other criteria contribute to decidability of constraint entailment. Criterion WF-IMPL-2, in combination with WF-PROG-4 as defined shortly, bears some resemblance to the *coverage condition* given by Sulzmann et al. [2007] for Haskell type classes. For criterion WF-IMPL-3, there exists a corresponding restriction in the Haskell 98 report [Peyton Jones 2003]. Sulzmann and colleagues' *bound-variable condition* [2007] is also similar to it.

Criteria for Programs. The notation $\Delta \vdash G_1 \sqcap G_2 = H$ denotes that H is the *greatest lower bound* of G_1 and G_2 with respect to Δ. Figure 16 defines this relation formally. The notation $\Delta \vdash \overline{G} \sqcap \overline{G'} = \overline{H}$ abbreviates $(\forall i)\ \Delta \vdash G_i \sqcap G'_i = H_i$.

The CoreGI program under consideration must fulfill the following well-formedness criteria:

WF-PROG-1 A program does not contain two different implementations for the same interface with unifiable implementing types. That is, for each pair of disjoint implementation definitions

$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{M}\,]\ \texttt{where}\ \overline{P}\ \ldots$$
$$\texttt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{Q}\ \ldots$$

it holds that, for all substitutions $[\overline{V/X}]$ and $[\overline{W/Y}]$, $[\overline{V/X}]\overline{M} \neq [\overline{W/Y}]\overline{N}$.

WF-PROG-2 A program does not contain two implementations of different instantiations of the same interface or for different non-dispatch types, provided the dispatch types of the implementations are subtype compatible. That is, for each pair of implementation definitions

$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{M}\,]\ \texttt{where}\ \overline{P}\ \ldots$$
$$\texttt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{Q}\ \ldots$$

and for all substitutions $[\overline{V/X}]$ and $[\overline{W/Y}]$ such that $\emptyset \vdash [\overline{V/X}]M_i \sqcap [\overline{W/Y}]N_i$ exists for all $i \in \mathsf{disp}(I)$, it holds that $[\overline{V/X}]\overline{T} = [\overline{W/Y}]\overline{U}$ and that $[\overline{V/X}]M_j = [\overline{W/Y}]N_j$ for all $j \notin \mathsf{disp}(I)$.

WF-PROG-3 Implementation definitions are downward closed. That is, for each pair of implementation definitions

$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{P}\ \ldots$$
$$\texttt{implementation}\langle\overline{X'}\rangle\ I\langle\overline{T'}\rangle\ [\,\overline{N'}\,]\ \texttt{where}\ \overline{P'}\ \ldots$$

and for all substitutions $[\overline{V/X}]$ and $[\overline{V'/X'}]$ with $\emptyset \vdash [\overline{V/X}]\overline{N} \sqcap [\overline{V'/X'}]\overline{N'} = \overline{M}$ there exists an implementation definition

$$\texttt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{M'}\,]\ \texttt{where}\ \overline{Q}\ \ldots$$

and a substitution $[\overline{W/Y}]$ such that $\overline{M} = [\overline{W/Y}]\overline{M'}$.

WF-PROG-4 Constraints on implementation definitions are consistent with constraints on implementation definitions for subclasses. That is, for each pair of implementation definitions

$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{M}\,]\ \texttt{where}\ \overline{P}\ \ldots$$
$$\texttt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{Q}\ \ldots$$

and for all substitutions $[\overline{V/X}]$ and $[\overline{W/Y}]$ with $[\overline{V/X}]\overline{M} \trianglelefteq_{\mathrm{c}} [\overline{W/Y}]\overline{N}$ and $\emptyset \Vdash [\overline{W/Y}]\overline{Q}$, it holds that $\emptyset \Vdash [\overline{V/X}]\overline{P}$.

WF-PROG-5 The class and interface graphs of the program are acyclic. (Each class definition $\texttt{class}\ C\langle\overline{X}\rangle\ \texttt{extends}\ D\langle\overline{T}\rangle\ \ldots$ contributes an edge $C \to D$ to the class graph, and each interface definition $\texttt{interface}\ I\langle\overline{X}\rangle\ [\overline{Y}\ \texttt{where}\ \overline{R}]\ \ldots$ and each constraint $\overline{G}\ \texttt{implements}\ J\langle\overline{V}\rangle \in \overline{R}$ contribute an edge $I \to J$ to the interface graph.)

WF-PROG-6 Multiple instantiation inheritance for interfaces is not permitted. That is, if $K \trianglelefteq_{\mathrm{i}} I\langle\overline{T}\rangle$ and $K \trianglelefteq_{\mathrm{i}} I\langle\overline{U}\rangle$ then $\overline{T} = \overline{U}$.

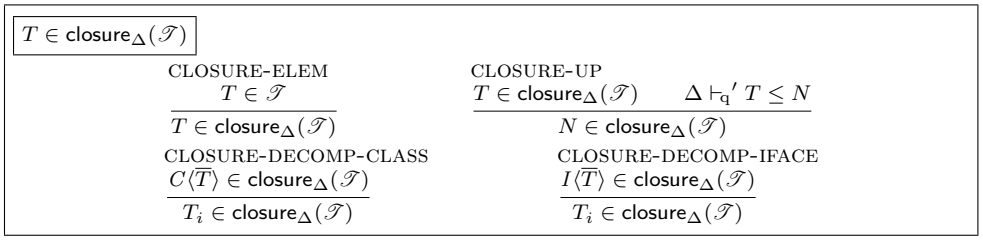

Figure 17: Closure of a set of types.

WF-PROG-7 Multiple inheritance for single-headed interfaces with neither positive nor negative polarity is not permitted. That is, if $1 \notin \mathsf{pol}^+(I)$, $1 \notin \mathsf{pol}^-(I)$, $I\langle\overline{T}\rangle \trianglelefteq_{\mathrm{i}} K_1$, and $I\langle\overline{T}\rangle \trianglelefteq_{\mathrm{i}} K_2$, then either $K_1 \trianglelefteq_{\mathrm{i}} K_2$ or $K_2 \trianglelefteq_{\mathrm{i}} K_1$.

We already discussed criteria WF-PROG-1 to WF-PROG-4 in Sections 3.4.1 to 3.4.4. Criteria WF-PROG-5 and WF-PROG-6 are standard for Java-like languages [Gosling et al. 2005, § 8.1.4, § 8.1.5, § 9.1.3]. The last criterion WF-PROG-7 is required to ensure that minimal types exist [Wehr 2010, pages 54-55].

Criteria for Type Environments. The following definition is due to Trifonov and Smith [1996].

Definition 4.7 *Contractive type environments.* A type environment Δ is *contractive* if, and only if, there exist no type variables $X_1, \ldots, X_n$ such that $X_1 = X_n$ and $X_i \,\mathtt{extends}\, X_{i+1} \in \Delta$ for each $i \in \{1, \ldots, n-1\}$.

The notation $\mathsf{closure}_\Delta(\mathscr{T})$ denotes the *closure* of a set of types $\mathscr{T}$ with respect to a type environment Δ. Figure 17 defines $\mathsf{closure}_\Delta(\mathscr{T})$ as the least superset of $\mathscr{T}$ closed under the kernel of quasi-algorithmic subtyping and under decomposition of generic class and interface types.

The well-formedness criteria on type environments now require that every type environment Δ must fulfill the following conditions.

WF-TENV-1 The type environment Δ is contractive.

WF-TENV-2 If $\mathscr{T}$ is a finite set of types, then the closure of $\mathscr{T}$ with respect to Δ is finite.

WF-TENV-3 A type variable does not have several unrelated G-types among its bounds. That is, if $X \,\mathtt{extends}\, G_1 \in \Delta$ and $X \,\mathtt{extends}\, G_2 \in \Delta$ then $\Delta \vdash G_1 \leq G_2$ or $\Delta \vdash G_2 \leq G_1$.

WF-TENV-4 A type variable is not a subtype of different instantiations of the same interface. That is, if $\Delta \vdash_{\mathrm{q}}' X \leq I\langle\overline{T}\rangle$ and $\Delta \vdash_{\mathrm{q}}' X \leq I\langle\overline{U}\rangle$ then $\overline{T} = \overline{U}$.

WF-TENV-5 A type variable has only negative interfaces among its bounds. That is, if $X \,\mathtt{extends}\, I\langle\overline{T}\rangle \in \Delta$ then $1 \in \mathsf{pol}^-(I)$.

WF-TENV-6 The type environment Δ does not contain two implementation constraints for different instantiations of the same interface or for different non-dispatch types in covariant position, provided the dispatch types of the implementation constraints are subtype compatible. The same holds for one implementation constraint in combination with an implementation definition. That is:

(1) For each pair of constraints
$$\overline{G}\,\texttt{implements}\,I\langle\overline{T}\rangle \in \mathsf{sup}(\Delta)$$
$$\overline{H}\,\texttt{implements}\,I\langle\overline{W}\rangle \in \mathsf{sup}(\Delta)$$
such that $\Delta \vdash G_i \sqcap H_i$ exists for all $i \in \mathsf{disp}(I)$, it holds that $\overline{T} = \overline{W}$ and $G_j = H_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$.

(2) For each constraint and each implementation definition
$$\overline{G}\,\texttt{implements}\,I\langle\overline{T}\rangle \in \mathsf{sup}(\Delta)$$
$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{W}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{P}\ \ldots$$
such that $\Delta \vdash G_i \sqcap [\overline{U/X}]N_i$ exists for all $i \in \mathsf{disp}(I)$ and some $\overline{U}$, it holds that $\overline{T} = [\overline{U/X}]\overline{W}$ and $G_j = [\overline{U/X}]N_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$.

Criterion WF-TENV-1 and WF-TENV-2 are required to establish decidability of constraint entailment and subtyping. Strictly speaking, criterion WF-TENV-2 is not compatible with JavaGI being a conservative extension of Java 1.5 because Java allows programs to have an infinitary closure of types. However, neither the authors nor other researchers are aware of any such programs with practical value [Viroli and Natali 2000; Kennedy and Pierce 2007]. Moreover, neither the Scala language [Odersky 2009, § 5.1.5] nor the Common Language Infrastructure of the .NET framework [ECMA International 2006, Partition II, § 9.2] allows programs to have an infinitary closure of types.

The well-formedness criteria WF-TENV-3, WF-TENV-4 (which is common for Java-like languages [Gosling et al. 2005, § 4.4]), WF-TENV-5, and WF-TENV-6 (which is somewhat related to WF-PROG-2) are all required to ensure existence of minimal types [Wehr 2010, pages 56–58].

4.5 Type Soundness and Determinacy of Evaluation

Having completed the definition of the static semantics, this sections proves that CoreGI enjoys type soundness and that its evaluation relation is deterministic. Moreover, the section shows that the declarative and the quasi-algorithmic formulations of constraint entailment and subtyping are equivalent. All intermediate lemmas and detailed proofs are available in the first author's dissertation [Wehr 2010]. The theorems presented in this section make the implicit assumption that the underlying CoreGI program is well-formed.

4.5.1 *Equivalence of Declarative and Quasi-algorithmic Constraint Entailment and Subtyping.* The type soundness proof relies on the equivalence of these two formulations of constraint entailment and subtyping.

THEOREM 4.8. *Quasi-algorithmic constraint entailment and subtyping are sound with respect to declarative constraint entailment and subtyping: (i) If* $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{R}$ *then* $\Delta \Vdash \mathcal{R}$. *(ii) If* $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$ *then* $\Delta \Vdash \mathcal{P}$. *(iii) If* $\Delta \vdash_{\mathrm{q}}{}' T \leq U$ *then* $\Delta \vdash T \leq U$. *(iv) If* $\Delta \vdash_{\mathrm{q}} T \leq U$ *then* $\Delta \vdash T \leq U$.

PROOF. The proof is by induction on the combined height of the derivations of $\Delta \Vdash_{\mathrm{q}}{}' \mathcal{R}$, $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$, $\Delta \vdash_{\mathrm{q}}{}' T \leq U$, and $\Delta \vdash_{\mathrm{q}} T \leq U$ [Wehr 2010, proof of Theorem 3.11, pages 177–179]. □

THEOREM 4.9. *Quasi-algorithmic constraint entailment and subtyping are complete with respect to declarative constraint entailment and subtyping: (i) If* $\Delta \Vdash \mathcal{P}$ *then* $\Delta \Vdash_{\mathrm{q}} \mathcal{P}$. *(ii) If* $\Delta \vdash T \leq U$ *then* $\Delta \vdash_{\mathrm{q}} T \leq U$.

PROOF. The proof is by induction on the combined height of the derivations of $\Delta \Vdash \mathcal{P}$ and $\Delta \vdash T \leq U$ [Wehr 2010, proof of Theorem 3.12, pages 179–197]. Well-formedness criterion WF-IMPL-1, which relies on quasi-algorithmic constraint entailment to ensure that superinterfaces are properly implemented, is crucial to the proof of claim (i). □

4.5.2 *Type Soundness.* The type soundness proof of CoreGI follows the syntactic approach pioneered by Wright and Felleisen [1994]. The progress theorem states that a well-typed expression is either a value or reduces to some other expression or is stuck on a bad cast.

Definition 4.10 *Stuck on a bad cast.* An expression e is *stuck on a bad cast* if, and only if, there exists an evaluation context $\mathcal{E}$, a type T, and a value $v = \mathtt{new}\, N(\overline{w})$ such that $e = \mathcal{E}[(T)\, v]$ and not $\emptyset \vdash N \leq T$.

THEOREM 4.11 PROGRESS. *If* $\emptyset; \emptyset \vdash e : T$ *then either* $e = v$ *for some value* v *or* $e \longrightarrow e'$ *for some expression* e' *or* e *is stuck on a bad cast.*

PROOF. The proof is by induction on the derivation of $\emptyset; \emptyset \vdash e : T$ [Wehr 2010, proof of Theorem 3.14, pages 198–210]. □

The preservation theorems for the evaluation relations $\longmapsto$ and $\longrightarrow$ show that evaluation of expressions preserves types.

THEOREM 4.12 PRESERVATION FOR TOP-LEVEL EVALUATION. *If* $\emptyset; \emptyset \vdash e : T$ *and* $e \longmapsto e'$ *then* $\emptyset; \emptyset \vdash e' : T$.

PROOF. The proof is by induction on the derivation of $\emptyset; \emptyset \vdash e : T$ [Wehr 2010, proof of Theorem 3.15, pages 210–228]. □

THEOREM 4.13 PRESERVATION FOR PROPER EVALUATION. *If* $\emptyset; \emptyset \vdash e : T$ *and* $e \longrightarrow e'$ *then* $\emptyset; \emptyset \vdash e' : T$.

PROOF. The derivation of $e \longrightarrow e'$ must end with rule DYN-CONTEXT, so there exists an evaluation context $\mathcal{E}$ and expressions e_0, e_0' such that $e = \mathcal{E}[e_0]$ and $e_0 \longmapsto e_0'$ and $\mathcal{E}[e_0'] = e'$. The claim $\emptyset; \emptyset \vdash \mathcal{E}[e'] : T$ now follows by induction on the structure of $\mathcal{E}$, using Theorem 4.12 for the base case. □

We let $\longrightarrow^*$ denote the reflexive, transitive closure of the evaluation relation $\longrightarrow$. The type soundness theorem for CoreGI is very similar to that for FGJ.

THEOREM 4.14 TYPE SOUNDNESS. *If* $\emptyset; \emptyset \vdash e : T$ *then either* e *diverges, or* $e \longrightarrow^* v$ *for some value* v *such that* $\emptyset; \emptyset \vdash v : T$, *or* $e \longrightarrow^* e'$ *for some expression* e' *such that* e' *is stuck on a bad cast.*

PROOF. Assume that $e \longrightarrow^* e'$ for some normal form e'. Theorem 4.13 and an induction on the length of the evaluation sequence yields $\emptyset; \emptyset \vdash e' : T$. The claim now follows by Theorem 4.11. □

A stronger type soundness theorem holds for programs without casts.

Definition 4.15 *Cast-free.* A program $\overline{def}\ e$ is *cast-free* if, and only if, neither e nor any method body in $\overline{def}$ contains a cast $(T)\,e'$ for some type T and some expression e'.

THEOREM 4.16 TYPE SOUNDNESS FOR PROGRAMS WITHOUT CASTS. *If $\overline{def}\ e$ is cast-free and $\emptyset;\emptyset \vdash e : T$ then either e diverges or $e \longrightarrow^* v$ for some value v such that $\emptyset;\emptyset \vdash v : T$.*

PROOF. If $e \longrightarrow^* e'$ and e is cast-free then so is e'. As a cast-free expression cannot be stuck on a bad cast, the claim follows with Theorem 4.14. □

4.5.3 *Determinacy of Evaluation.* CoreGI also enjoys a deterministic evaluation relation. This property is important because CoreGI's method lookup may involve more than one dispatch type, which could easily lead to ambiguities.

THEOREM 4.17 DETERMINACY OF EVALUATION. *If $e \longrightarrow e'$ and $e \longrightarrow e''$ then $e' = e''$.*

PROOF. We first show that least-impl, dynamic method lookup, and the top-level evaluation relation $\longmapsto$ are deterministic. Then we show that $\mathcal{E}_1[e_1] = \mathcal{E}_2[e_2]$, $e_1 \longmapsto e_1'$, and $e_2 \longmapsto e_2'$ imply $\mathcal{E}_1 = \mathcal{E}_2$. The claim now follows by inverting rule DYN-CONTEXT [Wehr 2010, proof of Theorem 3.20, pages 229–230]. □

4.6 Constraint Entailment and Subtyping Algorithms

The declarative specification of constraint entailment and subtyping in Section 4.2 is not immediately suitable for implementation: the conclusions of several rules overlap and the premises of rules ENT-SUPER, ENT-UP, and SUB-TRANS involve types not mentioned in the conclusions.

Section 4.4.3 introduced an equivalent, quasi-algorithmic formulation of entailment and subtyping. However, this formulation does not lead directly to an implementation either: the conclusions of several rules overlap, the premises of rules ENT-Q-ALG-UP and SUB-Q-ALG-IMPL involve types not present in the conclusions, and the recursive invocation of constraint entailment in rule ENT-Q-ALG-IMPL may lead to nontermination. To illustrate the danger of nontermination, consider the program in Figure 18. Searching for a derivation of $\emptyset \Vdash_{\mathrm{q}} D\,\texttt{implements}\,I$ leads to non-termination as demonstrated by the failed attempt in Figure 19.

Figure 20 shows an *algorithmic* formulation of constraint entailment and subtyping, which is easily amenable to implementation. It also contains a number of auxiliary algorithms as follows:

— Algorithmic constraint entailment, written $\Delta \Vdash_{\mathrm{a}} \mathcal{P}$, asserts validity of constraint $\mathcal{P}$ with respect to type environment Δ.

— The auxiliary relation $\Delta;\mathscr{G};\beta \Vdash_{\mathrm{a}} \mathcal{P}$ for algorithmic constraint entailment establishes validity of constraint $\mathcal{P}$ with respect to type environment Δ, goal cache $\mathscr{G}$, and boolean flag β. The goal cache $\mathscr{G}$ maintains the set of implementation constraints encountered while searching for a derivation. Rule ENT-ALG-IMPL avoids nontermination by performing recursive invocations only on constraints not contained in $\mathscr{G}$. The boolean flag β specifies whether type T_j of some constraint $\overline{T}\,\texttt{implements}\,I\langle\overline{V}\rangle$ may be lifted to a supertype without checking that the polarity of the jth implementing type of I is negative.

```
interface I [X] { receiver {} }
class C<X> extends Object {}
class D extends C<D> {}
implementation<X> I [C⟨X⟩] where X implements I { receiver {} }
```

Figure 18: Program demonstrating nontermination of quasi-algorithmic constraint entailment.

$$\dfrac{\dfrac{\text{(holds obviously)}}{\emptyset \vdash_{\mathrm{q}}{}' D \leq C\langle D\rangle} \quad \dfrac{\dfrac{\vdots}{\emptyset \Vdash_{\mathrm{q}} D\,\texttt{implements}\,I}}{\texttt{implementation}\langle X\rangle\ I\,[C\langle X\rangle]\ \texttt{where}\ X\ \texttt{implements}\ I\ \ldots}\ \text{ENT-Q-ALG-IMPL}}{\emptyset \Vdash_{\mathrm{q}} D\,\texttt{implements}\,I}\ \text{ENT-Q-ALG-UP}$$

$$\dfrac{\dfrac{\text{(holds obviously)}}{\emptyset \vdash_{\mathrm{q}}{}' D \leq C\langle D\rangle} \quad \dfrac{\texttt{implementation}\langle X\rangle\ I\,[C\langle X\rangle]\ \texttt{where}\ X\ \texttt{implements}\ I\ \ldots}{\emptyset \Vdash_{\mathrm{q}}{}' C\langle D\rangle\,\texttt{implements}\,I}\ \text{ENT-Q-ALG-IMPL}}{\emptyset \Vdash_{\mathrm{q}} D\,\texttt{implements}\,I}\ \text{ENT-Q-ALG-UP}$$

Figure 19: Failed attempt to construct a derivation of $\emptyset \Vdash_{\mathrm{q}} D\,\texttt{implements}\,I$.

— The auxiliary relation $\Delta;\beta;I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ lifts types $\overline{T}$ of some constraint $\overline{T}\,\texttt{implements}\,I\langle\overline{V}\rangle$ to supertypes $\overline{U}$ under type environment Δ. The role of β is the same as before.

— Algorithmic subtyping, written $\Delta \vdash_{\mathrm{a}} T \leq U$, states that T is a subtype of U under type environment Δ.

— The auxiliary relation $\Delta;\mathscr{G} \vdash_{\mathrm{a}} T \leq U$ states that T is a subtype of U under type environment Δ and goal cache $\mathscr{G}$. Rule SUB-ALG-KERNEL falls back to the kernel variant of quasi-algorithmic subtyping because the corresponding rules are already syntax-directed and easily implementable (see Figure 14).

Following the rules in Figure 20 and the rules for quasi-algorithmic kernel subtyping in Figure 14, the implementation of an entailment and subtype checker becomes straightforward [Wehr 2010, Figures B.3 and B.4]. Only two details need further explanation:

—Rules ENT-ALG-ENV, ENT-ALG-IFACE$_1$, ENT-ALG-IFACE$_2$, and ENT-ALG-IMPL overlap. The implementation simply tries the rules in order of their appearance until one succeeds or all fail.

—Rule ENT-ALG-IMPL lifts types $\overline{T}$ to class types $[\overline{U/X}]\overline{N}$, which requires finding a suitable substitution $[\overline{U/X}]$. In other words, $[\overline{U/X}]$ must solve the matching problem modulo kernel subtyping $(\Delta, \overline{X}, \{T_1 \leq^? N_1, \ldots, T_n \leq^? N_n\})$.

Matching modulo kernel subtyping is a special case of unification modulo kernel subtyping, which the forthcoming Section 4.8 needs anyway. In the following, the notation $\mathsf{ftv}(\Delta)$ denotes the set $\bigcup\{\mathsf{ftv}(P) \mid P \in \Delta\}$ for some type environment Δ.

Definition 4.18 *Unification modulo kernel subtyping.* A *unification problem modulo kernel subtyping* is a triple $\mathbb{U} = \big(\Delta, \overline{X}, \{T_1 \leq^? U_1, \ldots T_n \leq^? U_n\}\big)$ such that $\mathsf{ftv}(\Delta) \cap \overline{X} = \emptyset$ and $T_i = Y$ (or $U_i = Y$) implies $Y \notin \overline{X}$ for all $i \in [n]$. A *solution* of $\mathbb{U}$ is a substitution $\varphi = [\overline{V/X}]$ such that $\Delta \vdash_{\mathrm{q}}{}' \varphi T_i \leq \varphi U_i$ for all $i = 1, \ldots, n$. A *most general solution* of $\mathbb{U}$ is a solution φ that is more general than any other

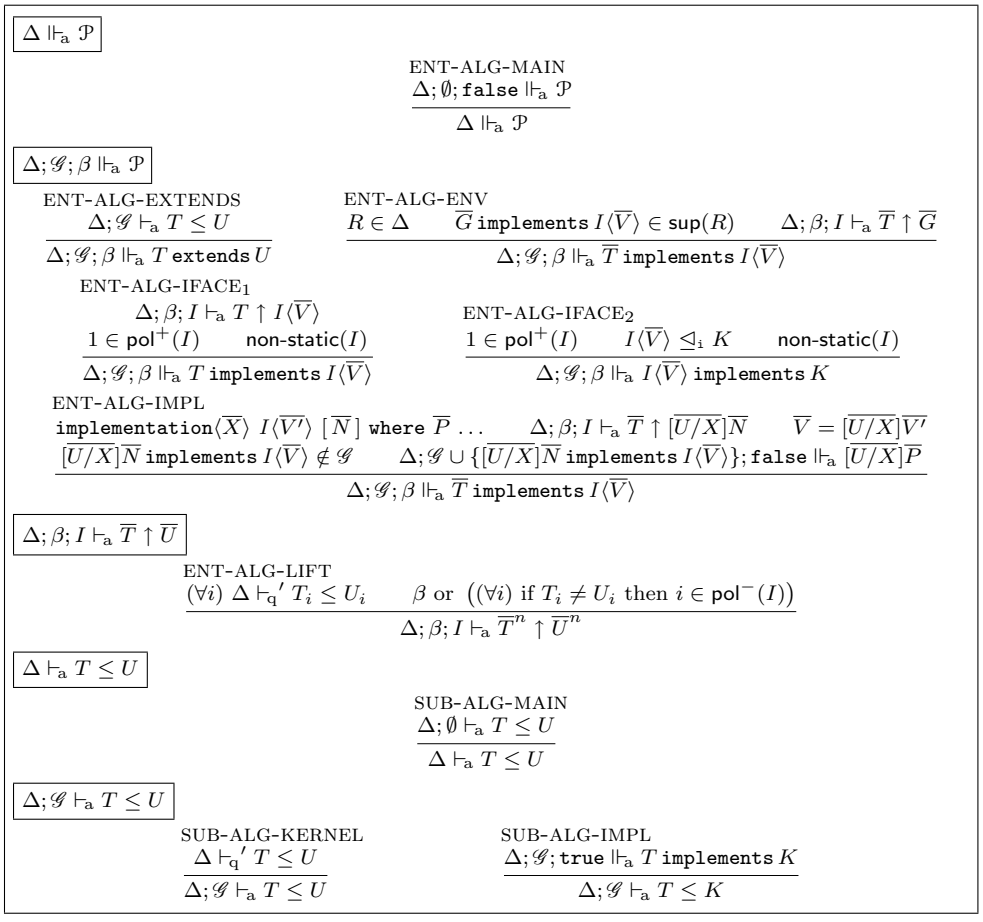

Figure 20: Algorithmic constraint entailment and subtyping.

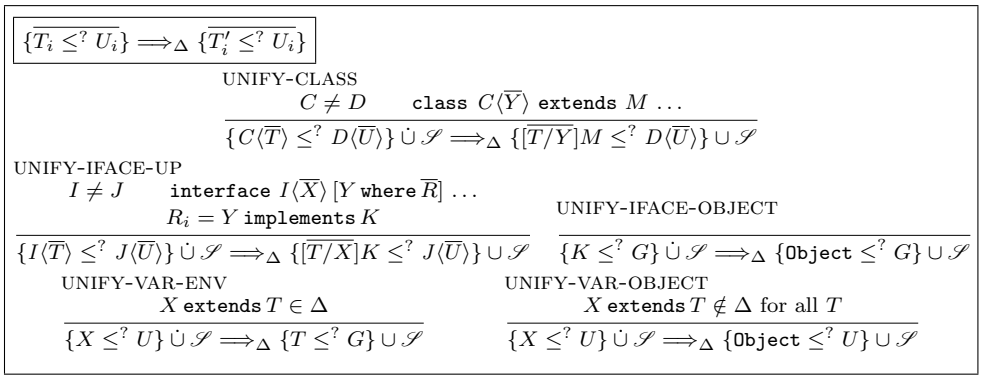

Figure 21: Transformation of unification modulo kernel subtyping problems.

solution φ' of $\mathbb{U}$; that is, there exists a substitution ψ such that $\varphi' = \psi\varphi$ (where $\psi\varphi$ denotes the composition of ψ and φ).

The relation $\{\overline{T_i \leq^? U_i}\} \Longrightarrow_\Delta \{\overline{T_i' \leq^? U_i}\}$, defined in Figure 21, transforms a set of inequations $\{\overline{T_i \leq^? U_i}\}$ into $\{\overline{T_i' \leq^? U_i}\}$ by lifting one of the types T_i to a direct supertype T_i' under type environment Δ. The notation $\mathscr{M}_1 \dot\cup \mathscr{M}_2$ denotes the disjoint union of $\mathscr{M}_1$ and $\mathscr{M}_2$.

Definition 4.19 *Algorithm for solving unification modulo kernel subtyping.* The procedure $\mathtt{unify}_\leq(\mathbb{U})$ solves a unification problem modulo kernel subtyping $\mathbb{U} = (\Delta, \overline{X}, \mathscr{S})$ by first reducing $\mathscr{S}$ to all its normal forms with respect to $\Longrightarrow_\Delta$. If syntactic unification [Baader and Nipkow 1998] succeeds for any of these normal forms and returns a solution φ, $\mathtt{unify}_\leq(\mathbb{U})$ also returns φ. Otherwise, it fails.

THEOREM 4.20 SOUNDNESS AND COMPLETENESS OF $\mathtt{unify}_\leq$. *Let $\mathbb{U}$ be a unification problem modulo kernel subtyping. If $\mathtt{unify}_\leq(\mathbb{U})$ returns a substitution φ then φ is an idempotent, most general solution of $\mathbb{U}$ (soundness). Moreover, if $\mathbb{U}$ has a solution, then $\mathtt{unify}_\leq(\mathbb{U})$ does not fail (completeness).*

PROOF. If $\mathbb{U} = (\Delta, \overline{X}, \mathscr{S})$ and $\mathscr{S} \Longrightarrow_\Delta \mathscr{S}'$ then $(\Delta, \overline{X}, \mathscr{S}')$ is a unification problem modulo kernel subtyping with the same solution set as $\mathbb{U}$. The claim now follows because type constructors are invariant and because syntactic unification is sound and complete [Baader and Nipkow 1998]. □

THEOREM 4.21 TERMINATION OF $\mathtt{unify}_\leq$. *Let $\mathbb{U}$ be a unification problem modulo kernel subtyping. Then $\mathtt{unify}_\leq(\mathbb{U})$ terminates.*

PROOF. Holds because syntactic unification terminates [Baader and Nipkow 1998] and the reduction relation $\Longrightarrow$ is terminating [Wehr 2010, proof of Theorem 3.24, pages 230–231]. □

Equivalence of algorithmic and quasi-algorithmic entailment and subtyping follows with the next two theorems.

THEOREM 4.22. *Algorithmic constraint entailment and subtyping are sound with respect to quasi-algorithmic constraint entailment and subtyping: (i) If $\Delta \Vdash_\mathrm{a} \mathcal{P}$ then $\Delta \Vdash_\mathrm{q} \mathcal{P}$. (ii) If $\Delta \vdash_\mathrm{a} T \leq U$ then $\Delta \vdash_\mathrm{q} T \leq U$*

PROOF. See [Wehr 2010, proof of Theorem 3.25, pages 231–233]. □

THEOREM 4.23. *Algorithmic constraint entailment and subtyping are complete with respect to quasi-algorithmic constraint entailment and subtyping. (i) If $\Delta \Vdash_\mathrm{q} \mathcal{P}$ then $\Delta \Vdash_\mathrm{a} \mathcal{P}$. (ii) If $\Delta \vdash_\mathrm{q} T \leq U$ then $\Delta \vdash_\mathrm{a} T \leq U$.*

PROOF. See [Wehr 2010, proof of Theorem 3.26, pages 233–237]. □

Equivalence between the algorithmic and the declarative formulations of constraint entailment and subtyping then follows with Theorems 4.8 and 4.9. Algorithmic constraint entailment and subtyping also terminates:

THEOREM 4.24. *The entailment and subtyping algorithms induced by the rules in Figure 20 and by the rules for quasi-algorithmic kernel subtyping in Figure 14 terminate.*

PROOF. The proof relies on well-formedness criterion WF-TENV-2 to show that the goal cache $\mathscr{G}$ does not grow indefinitely [Wehr 2010, proof of Theorem 3.27, pages 237–246]. □

4.7 Expression Typing Algorithm

The declarative specification of the typing relation for expressions (see Section 4.4.1) is not suited for implementing a typechecking algorithm. The main culprit is the explicit subsumption rule EXP-SUBSUME that permits lifting the type of an expression to some arbitrary supertype. This section presents a syntax-directed variant of expression typing that is suitable for implementation and that computes minimal types.

4.7.1 *Algorithmic Method Typing.* Algorithmic method typing compensates for the lack of an explicit subsumption rule in the syntax-directed variant of expression typing (to be defined shortly) by integrating subsumption into method typing. Furthermore, it infers those types which the declarative specification of method typing must guess. Consider rule MTYPE-IFACE from Figure 9. An application of this rule must guess all types T_i for $i \neq j$ and all types $\overline{V}$. Even if mtype also had access to the types of the actual parameters of a method invocation, this would, in general, not be enough to determine all $\overline{T}$ and all $\overline{V}$.

Fortunately, well-formedness criteria WF-PROG-2 and WF-TENV-6 make it possible to define an algorithmic variant of mtype that infers those $\overline{T}$ and $\overline{V}$ that are needed to compute the type (i.e., signature) of a method. Figure 22 defines the first part of the inference machinery by extending algorithmic constraint entailment to *entailment for constraints with optional types*.

A *constraint with optional types* has the form $\overline{T^?}\,\texttt{implements}\,I\langle\overline{U^?}\rangle$, where each $T_i^?$ and each $U_i^?$ is optional (i.e., either nil or a regular type). Entailment for such constraints has the form $\Delta \Vdash_{\mathrm{a}}^? \overline{T^?}\,\texttt{implements}\,I\langle\overline{U^?}\rangle \twoheadrightarrow \overline{T}\,\texttt{implements}\,I\langle\overline{U}\rangle$. It completes $\overline{T^?}\,\texttt{implements}\,I\langle\overline{U^?}\rangle$ to the constraint $\overline{T}\,\texttt{implements}\,I\langle\overline{U}\rangle$ by inferring types for those $T_i^?$ and $U_i^?$ that are nil. Moreover, it ensures that the completed constraint $\overline{T}\,\texttt{implements}\,I\langle\overline{U}\rangle$ holds under type environment Δ. The definition of algorithmic entailment for constraint with optional types relies on several auxiliaries:

—The auxiliary $\Delta;\mathscr{G};\beta \Vdash_{\mathrm{a}}^? \overline{T^?}\,\texttt{implements}\,I\langle\overline{U^?}\rangle \twoheadrightarrow \overline{T}\,\texttt{implements}\,I\langle\overline{U}\rangle$ is analogous to $\Delta;\mathscr{G};\beta \Vdash_{\mathrm{a}} \overline{T}\,\texttt{implements}\,\overline{U}$ from Figure 20.

—The auxiliary $\Delta;\beta;I \vdash_{\mathrm{a}}^? \overline{T^?} \uparrow \overline{U} \twoheadrightarrow \overline{T}$ is analogous to $\Delta;\beta;I \vdash_{\mathrm{a}} \overline{T} \uparrow \overline{U}$ from Figure 20: it lifts those $T_i^? \neq \mathsf{nil}$ to a supertype U_i and completes those $T_i^? = \mathsf{nil}$ to U_i.

—The auxiliary $T^? \sim T$ matches an optional type $T^?$ with a regular type T.

THEOREM 4.25. *Entailment for constraints with optional types is sound with respect to algorithmic entailment: If* $\Delta \Vdash_{\mathrm{a}}^? \overline{T^?}\,\texttt{implements}\,I\langle\overline{W^?}\rangle \twoheadrightarrow \mathcal{R}$ *then* $\Delta \Vdash_{\mathrm{a}} \mathcal{R}$.

PROOF. The proof is by induction on the derivation given [Wehr 2010, proof of Theorem 3.28, pages 246–248]. □

THEOREM 4.26. *Entailment for constraints with optional types is complete with respect to algorithmic entailment: If* $\Delta \Vdash_{\mathrm{a}} \overline{T}\,\texttt{implements}\,I\langle\overline{V}\rangle$ *and* $\overline{T^?}\,\overline{V^?} \sim \overline{T}\,\overline{V}$

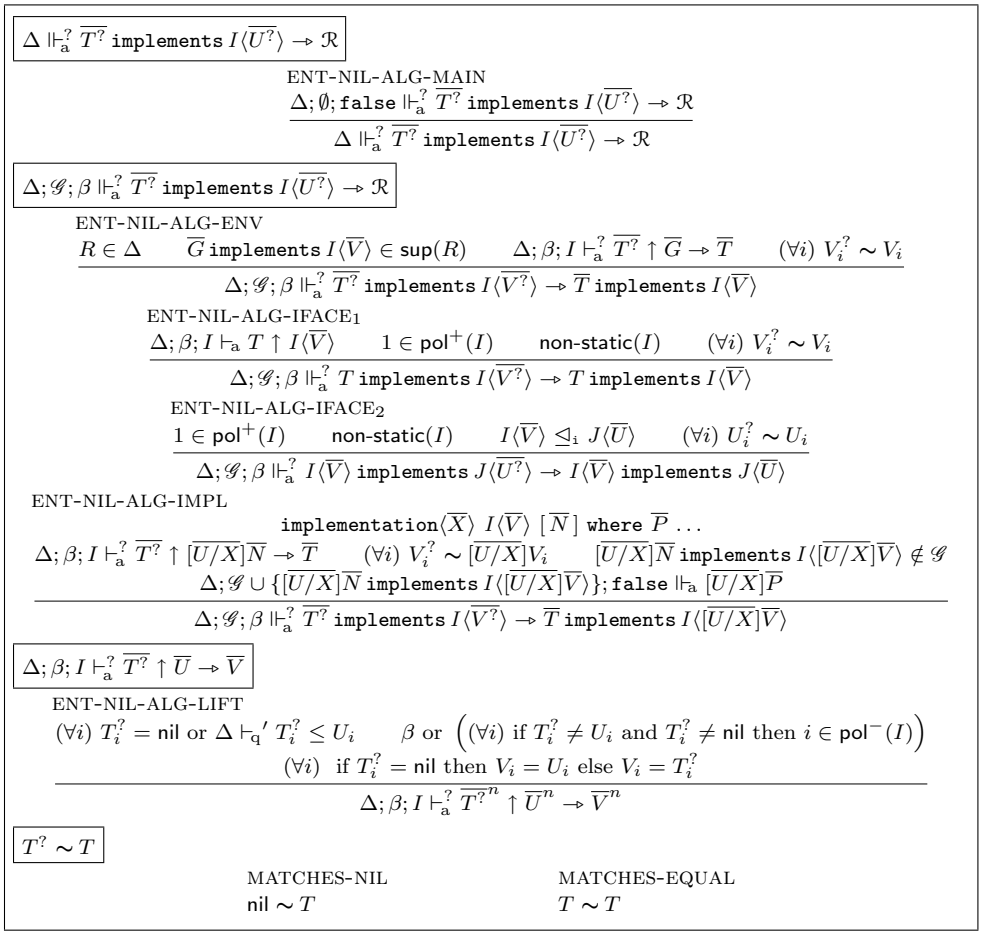

Figure 22: Entailment for constraints with optional types.

and $T_i^{?} \neq \mathsf{nil}$ *for* $i \in \mathsf{disp}(I)$*, then* $\Delta \Vdash_{\mathrm{a}}^{?} \overline{T^{?}}\ \mathtt{implements}\ I\langle\overline{V^{?}}\rangle \twoheadrightarrow \overline{U}\ \mathtt{implements}\ I\langle\overline{V}\rangle$ *such that* $\Delta \vdash_{\mathrm{q}}{}' T_i \leq U_i$ *for all* i *and* $U_i = T_i$ *for those* i *with* $T_i^{?} \neq \mathsf{nil}$ *or* $i \notin \mathsf{pol}^{-}(I)$*.*

Proof. The claim follows with a case distinction on the last rule of the derivation given [Wehr 2010, proof of Theorem 3.29, pages 248–249] □

Figure 23 formalizes algorithmic method typing. The relation $\mathsf{a\text{-}mtype}_{\Delta}(m, T, \overline{T})$ determines the signature of non-static method m when invoked on receiver and arguments with static types T and $\overline{T}$, respectively. The relation $\mathsf{a\text{-}smtype}_{\Delta}(m, I\langle\overline{U}\rangle[\overline{T}])$ determines the signature of a static interface method m for interface $I\langle\overline{U}\rangle$ and implementing types $\overline{T}$. The definition of $\mathsf{a\text{-}smtype}$ is straightforward, the one for $\mathsf{a\text{-}mtype}$ requires several auxiliaries, which Figure 23 also defines:

— $\mathsf{a\text{-}mtype}^{\mathrm{c}}(m, N) = \langle\overline{X}\rangle\, \overline{U\, x} \to U\ \mathtt{where}\ \overline{\mathcal{P}}$ determines the signature of a class method m by ascending the inheritance hierarchy starting at class N. The $\mathsf{a\text{-}mtype}^{\mathrm{c}}$ relation is very similar to the method typing relation of Featherweight Generic Java [Igarashi et al. 2001].

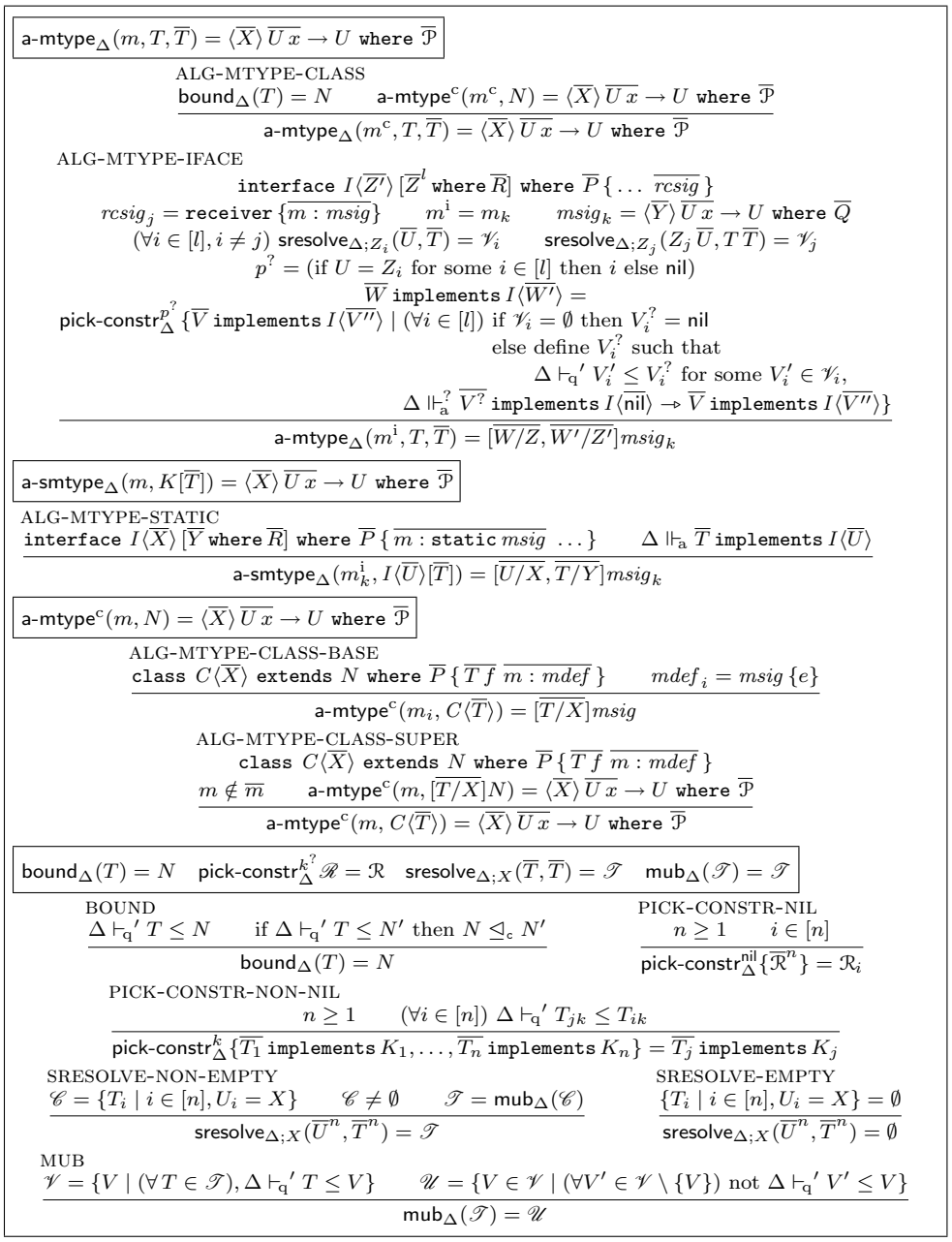

Figure 23: Algorithmic method typing.

— $\mathsf{bound}_\Delta(T) = N$ computes the bound N of a type T with respect to a type environment Δ.

— $\mathsf{pick\text{-}constr}_\Delta^{k^?}\mathscr{R} = \mathcal{R}$ takes a set $\mathscr{R}$ of $\mathcal{R}$-constraints, a type environment Δ, and an optional index $k^?$. If $k = \mathsf{nil}$ and $\mathscr{R} \neq \emptyset$, pick-constr returns an arbitrary constraint $\mathcal{R} \in \mathscr{R}$. If $k \in \mathbb{N}$ and $\mathscr{R} \neq \emptyset$, it returns a constraint $\mathcal{R} \in \mathscr{R}$ such that the kth implementing type of $\mathcal{R}$ is minimal with respect to the kth implementing types of all other constraints in $\mathscr{R}$.

— $\mathsf{sresolve}_{\Delta;X}(\overline{U}, \overline{T}) = \mathscr{T}$ is the static counterpart resolve from Figure 6. It resolves implementing type X with respect to formal parameter types $\overline{U}$, the static types $\overline{T}$ of the actual parameters, and type environment Δ. Whereas resolve returns an optional type (the least upper bound, if existing, of a set of class types), sresolve returns a set of types (the minimal elements of the upper bounds of all static parameter types contributing to the resolution of X).

— $\mathsf{mub}_\Delta(\mathscr{T}) = \mathscr{U}$ takes a set of types $\mathscr{T}$ and returns a set of types $\mathscr{U}$ containing the minimal elements of the upper bounds of all types in $\mathscr{T}$.

The definition of a-mtype for class methods relies on $\mathsf{a\text{-}mtype}^\mathsf{c}$ to find the signature of the method in question. The definition of a-mtype for interface methods is more involved:

—First, a-mtype retrieves interface I and receiver $rcsig_j$ defining method m.

—Then, it uses sresolve to compute, for each implementing type variable Z_i, a set $\mathscr{V}_i$. This set contains the minimal elements of the upper bounds of all static argument types that contribute to the resolution of the ith implementing type.

—Next, it collects all implementation constraints for I that are entailed by Δ and that match the $\mathscr{V}_i$ pointwise. This step also infers unknown types.

—Finally, a-mtype uses $\mathsf{pick\text{-}constr}_\Delta^{p^?}$ to pick an element from the collected constraints. To minimize the result type of the signature computed by a-mtype, $p^? \neq \mathsf{nil}$ if, and only if, the signature declared in the interface uses the pth implementing type as its result type. (Criterion WF-IFACE-3 ensures that implementing types do not occur nested inside the result type.)

Definition 4.27. A type environment Δ is well-formed, written $\vdash \Delta\ \mathsf{ok}$ if, and only if, $\Delta \vdash P\ \mathsf{ok}$ for all $P \in \Delta$.

THEOREM 4.28 SOUNDNESS OF ALGORITHMIC METHOD TYPING. *Assume that* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash T, \overline{T}\ \mathsf{ok}$. *If* $\mathsf{a\text{-}mtype}_\Delta(m, T, \overline{T}) = \langle \overline{X} \rangle\, \overline{U\, x} \to U\ \mathtt{where}\ \overline{\mathcal{P}}$ *then there exists a type* T' *such that* $\Delta \vdash T \leq T'$ *and* $\mathsf{mtype}_\Delta(m, T') = \langle \overline{X} \rangle\, \overline{U\, x} \to U\ \mathtt{where}\ \overline{\mathcal{P}}$.

PROOF. See [Wehr 2010, proof of Theorem 3.31, pages 249–250]. □

THEOREM 4.29 COMPLETENESS OF ALGORITHMIC METHOD TYPING. *Assume* $\mathsf{mtype}_\Delta(m, T) = \langle \overline{X} \rangle\, \overline{U\, x}^n \to U\ \mathtt{where}\ \overline{\mathcal{P}}$ *and let* φ *be a substitution* $[\overline{V/X}]$. *Furthermore, suppose* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash T'\ \mathsf{ok}$. *If* $\Delta \vdash T' \leq T$ *and* $\Delta \vdash T_i \leq \varphi U_i$ *for all* $i \in [n]$ *and* $\Delta \Vdash \varphi\overline{\mathcal{P}}$, *then* $\mathsf{a\text{-}mtype}_\Delta(m, T', \overline{T}) = \langle \overline{X} \rangle\, \overline{U'\, x}^n \to U'\ \mathtt{where}\ \overline{\mathcal{P}}$ *such that* $\Delta \vdash T_i \leq \varphi U_i'$ *for all* $i \in [n]$ *and* $\Delta \vdash \varphi U' \leq \varphi U$.

PROOF. See [Wehr 2010, proof of Theorem 3.32, pages 251–271] □

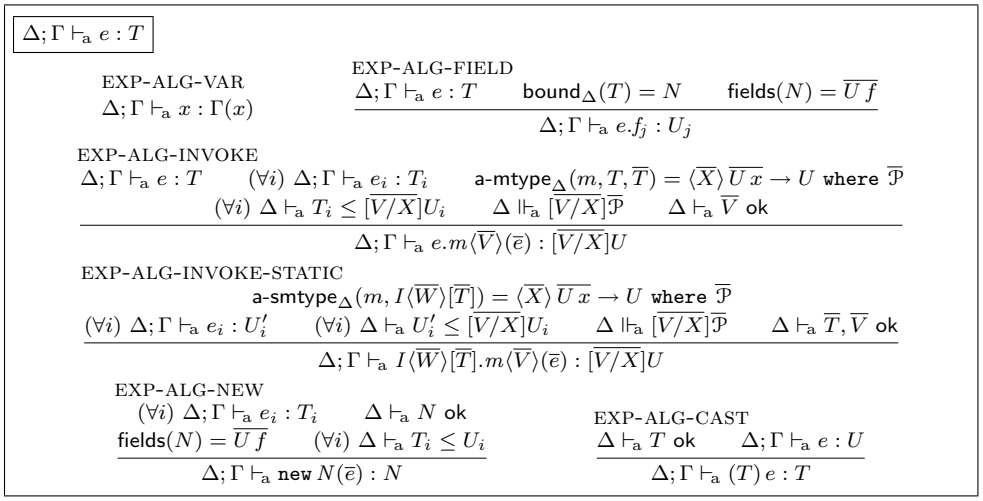

Figure 24: Algorithmic expression typing.

4.7.2 *Algorithmic Expression Typing.* With algorithmic method typing in hand, the definition of an algorithm for typechecking expressions is straightforward and follows closely the approach taken by Featherweight Generic Java [Igarashi et al. 2001]. Figure 24 presents the relation $\Delta;\Gamma \vdash_{\mathrm{a}} e : T$ that assigns type T to expression e under type environment Δ and variable environment Γ. The rules defining the relation are syntax-directed and easy to implement. They rely on algorithmic formulations of the well-formedness judgments from Figure 8:

Definition 4.30. The relations $\Delta \vdash_{\mathrm{a}} T\ \mathsf{ok}$ and $\Delta \vdash_{\mathrm{a}} \mathcal{P}\ \mathsf{ok}$ are defined analogously to the relations $\Delta \vdash T\ \mathsf{ok}$ and $\Delta \vdash \mathcal{P}\ \mathsf{ok}$, respectively, replacing $\vdash$ with $\vdash_{\mathrm{a}}$ and $\Vdash$ with $\Vdash_{\mathrm{a}}$.

Algorithmic expression typing is equivalent to the declarative specification of expression typing in Figure 10.

Definition 4.31. A variable environment Γ is well-formed under type environment Δ, written $\Delta \vdash \Gamma\ \mathsf{ok}$, if, and only if, $\Delta \vdash T\ \mathsf{ok}$ for all $x : T$ in Γ.

Theorem 4.32 Soundness of algorithmic expression typing. *Suppose* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash \Gamma\ \mathsf{ok}$. *If* $\Delta;\Gamma \vdash_{\mathrm{a}} e : T$ *then* $\Delta;\Gamma \vdash e : T$.

Proof. The proof is by induction on the derivation of $\Delta;\Gamma \vdash_{\mathrm{a}} e : T$ [Wehr 2010, proof of Theorem 3.35, pages 271–281]. □

Theorem 4.33 Completeness of algorithmic expression typing. *Assume* $\vdash \Delta\ \mathsf{ok}$ *and* $\Delta \vdash \Gamma\ \mathsf{ok}$. *If* $\Delta;\Gamma \vdash e : T$ *then* $\Delta;\Gamma \vdash_{\mathrm{a}} e : U$ *such that* $\Delta \vdash U \le T$.

Proof. The proof is by induction on the derivation of $\Delta;\Gamma \vdash e : T$ [Wehr 2010, proof of Theorem 3.36, pages 281–283]. □

Algorithmic expression typing also terminates.

Theorem 4.34. *The algorithm induced by the rules in Figures 22, 23, and 24 terminates.*

PROOF. See [Wehr 2010, proof of Theorem 3.37, pages 283–284]. □

4.8 Program Typing Algorithm

Given the algorithms for constraint entailment, subtyping, and expression typing, implementing a typechecker for CoreGI programs is almost straightforward, only the implementation of well-formedness criteria WF-PROG-2, WF-PROG-3, WF-PROG-4, WF-TENV-2, and WF-TENV-6(2) poses a challenge.

4.8.1 *Checking* WF-PROG-2*,* WF-PROG-3*,* WF-PROG-4*, and* WF-TENV-6(2). A direct implementation of these criteria is not possible because their definition involves universal quantification over substitutions subject to subtype or greatest lower bound conditions.

Definition 4.35 *Unification modulo greatest lower bounds.* A *unification problem modulo greatest lower bounds* is a triple $\mathbb{L} = \big(\Delta, \overline{X}, \{G_1 \sqcap^? H_1, \dots G_n \sqcap^? H_n\}\big)$ such that $\mathsf{ftv}(\Delta) \cap \overline{X} = \emptyset$ and $G_i = Y$ (or $H_i = Y$) implies $Y \notin \overline{X}$ for all $i \in [n]$. A *solution* of $\mathbb{L}$ is a substitution $\varphi = [\overline{V/X}]$ such that $\Delta \vdash \varphi G_i \sqcap \varphi H_i$ exists for all $i = 1, \dots, n$. A *most general solution* of $\mathbb{L}$ is a solution that is more general than any other solution of $\mathbb{L}$ (see Definition 4.18).

Obviously, φ solves $(\Delta, \overline{X}, \{G_{11} \sqcap^? G_{12}, \dots, G_{n1} \sqcap^? G_{n2}\})$ if, and only if, it also solves the the unification problem modulo kernel subtyping $(\Delta, \overline{X}, \{G_{1i_1} \leq^? G_{1j_1}, \dots, G_{ni_n} \leq^? G_{nj_n}\})$ for some set of pairs $\{(i_1, j_1), \dots, (i_n, j_n)\}$ where $(i_k, j_k) \in \{(1,2),(2,1)\}$ for all $k \in [n]$. Thus, a naive algorithm for solving unification problems modulo greatest lower bounds simply enumerates all of these unification problems modulo kernel subtyping and checks whether any of them has a solution φ. If so, it returns φ and fails otherwise. Call this naive algorithm with complexity $\Omega(2^n)$ $\mathtt{unify}_\sqcap$.

THEOREM 4.36 SOUNDNESS, COMPLETENESS, AND TERMINATION OF $\mathtt{unify}_\sqcap$. *Let $\mathbb{L}$ be a unification problem modulo greatest lower bounds. If $\mathbb{L}$ has a solution then $\mathtt{unify}_\sqcap(\mathbb{L})$ returns an idempotent, most general solution of $\mathbb{L}$. If $\mathbb{L}$ does not have a solution, $\mathtt{unify}_\sqcap(\mathbb{L})$ terminates with a failure.*

PROOF. See [Wehr 2010, proof of Theorem 3.39, pages 285–286]. □

The following alternative formulations of WF-PROG-2, WF-PROG-3, WF-PROG-4, and WF-TENV-6(2) are straightforward to implement.

WF-PROG-2′ For each pair of disjoint implementation definitions

$$\mathtt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{M}\,]\ \mathtt{where}\ \overline{P}\ \dots$$
$$\mathtt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{N}\,]\ \mathtt{where}\ \overline{Q}\ \dots$$

with $\overline{X} \cap \overline{Y} = \emptyset$ and where $\mathtt{unify}_\sqcap(\emptyset, \overline{X}\,\overline{Y}, \{M_i \sqcap^? N_i \mid i \in \mathsf{disp}(I)\}) = \varphi$, it holds that $\varphi\overline{T} = \varphi\overline{U}$ and that $\varphi M_j = \varphi N_j$ for all $j \notin \mathsf{disp}(I)$.

WF-PROG-3′ For each pair of disjoint implementation definitions

$$\mathtt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{N}^n\,]\ \mathtt{where}\ \overline{P}\ \dots$$
$$\mathtt{implementation}\langle\overline{X'}\rangle\ I\langle\overline{T'}\rangle\ [\,\overline{N'}^n\,]\ \mathtt{where}\ \overline{P'}\ \dots$$

with $\overline{X} \cap \overline{X'} = \emptyset$ and where $\mathtt{unify}_\sqcap(\emptyset, \overline{X}\,\overline{X'}, \{N_i \sqcap^? N'_i \mid i \in [n]\}) = \varphi$, there exists an implementation definition

$$\mathtt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{M}\,]\ \mathtt{where}\ \overline{Q}\ \dots$$

and a substitution $[\overline{W/Y}]$ such that $\emptyset \vdash \varphi\overline{N} \sqcap \varphi\overline{N'} = [\overline{W/Y}]\overline{M}$.

WF-PROG-4′ For each pair of disjoint implementation definitions

$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{T}\rangle\ [\,\overline{M}\,]\ \texttt{where}\ \overline{P}\ \ldots$$
$$\texttt{implementation}\langle\overline{Y}\rangle\ I\langle\overline{U}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{Q}\ \ldots$$

with $\overline{X} \cap \overline{X'} = \emptyset$ and where $\texttt{unify}_{\leq}(\emptyset, \overline{X}\,\overline{Y}, \{M_i \leq^? N_i \mid i \in [n]\}) = \varphi$, it holds that for all $\mathcal{P} \in \varphi\overline{P}$ either $\{Q \in \varphi\overline{Q}\} \Vdash \mathcal{P}$ or $\mathcal{P} \in \varphi\overline{Q} \cup \mathsf{sup}(\varphi\overline{Q}) \cup \{T\ \texttt{extends}\ U \mid T\ \texttt{extends}\ U' \in \varphi\overline{Q}, \{Q \in \varphi\overline{Q}\} \vdash_{\mathrm{q}}{}' U' \leq U\}$.

WF-TENV-6′

(1) Unchanged from criterion WF-TENV-6.

(2) For each constraint and each implementation definition

$$\overline{G}\,\texttt{implements}\,I\langle\overline{T}\rangle \in \mathsf{sup}(\Delta)$$
$$\texttt{implementation}\langle\overline{X}\rangle\ I\langle\overline{W}\rangle\ [\,\overline{N}\,]\ \texttt{where}\ \overline{P}\ \ldots$$

with $\overline{X} \cap (\bigcup\{\mathsf{ftv}(\mathcal{S}) \mid \mathcal{R} \in \Delta, \mathcal{S} \in \mathsf{sup}(\mathcal{R})\}) = \emptyset$ and where $\texttt{unify}_{\sqcap}(\Delta, \overline{X}, \{G_i \sqcap^? N_i \mid i \in \mathsf{disp}(I)\}) = \varphi$, it holds that $\overline{T} = \varphi\overline{W}$ and $G_j = \varphi N_j$ for all $j \notin \mathsf{disp}(I) \cup \mathsf{pol}^-(I)$.

THEOREM 4.37. *Criteria* WF-PROG-2′, WF-PROG-3′, *and* WF-TENV-6′ *are equivalent to their counterparts from Section 4.4.3. Criterion* WF-PROG-4′ *is sound with respect to* WF-PROG-4 *(i.e.,* WF-PROG-4′ *implies* WF-PROG-4*).*

PROOF. See [Wehr 2010, proof of Theorem 3.40, pages 286–287]. □

It is an open question whether there exists a complete algorithm for checking well-formedness criterion WF-PROG-4.

4.8.2 *Checking* WF-TENV-2. This criterion requires the closure of a finite set of types to be finite. Thanks to Viroli [2000], there is an equivalent but syntactic characterization of this property. Roughly speaking, Viroli's approach defines a dependency graph between the formal type parameters of all classes such that finitary closure of a finite set of types is equivalent to the absence of certain cycles in the dependency graph. Recasting Viroli's approach in the context of CoreGI leads to an equivalent and implementable formulation of well-formedness criterion WF-TENV-2 [Wehr 2010, Appendix B.7].

Concluding Remarks. This section formalized CoreGI, a small calculus capturing most aspects of the generalized interface mechanism of JavaGI. The formalization included the definition of CoreGI's dynamic semantics and a declarative specification of its static semantics. Two important properties hold for a well-typed CoreGI program: evaluation is deterministic and evaluation cannot get stuck if all cast operations succeed.

Besides proving these properties, the section also demonstrated how to typecheck CoreGI programs. To this end, algorithmic formulations of constraint entailment, subtyping, method typing, expression typing, and program typing were presented.

5. IMPLEMENTATION

The JavaGI compiler is an extension of the Eclipse Compiler for Java [Eclipse Foundation 2008] and generates byte code that runs on a standard Java Virtual Machine

(JVM [Lindholm and Yellin 1999]). Besides the plain compiler and an accompanying run-time system, there also exists IDE support for JavaGI in form of an Eclipse [2009] plugin. The homepage of the JavaGI project [Wehr 2009] makes the source code of the compiler, the run-time system, and the Eclipse plugin available under the terms of the Eclipse Public License [Eclipse Foundation 2004].

This section sketches how to extend CoreGI to the full JavaGI language. Moreover, it discusses the translation of JavaGI constructs to Java byte code and describes JavaGI's run-time system.

5.1 Extending CoreGI to JavaGI

The CoreGI calculus from Section 4 lacks several features present in the full JavaGI language. Section 3.3 already sketched how to typecheck method invocations without CoreGI's restrictions that identifier sets for class and interface methods are disjoint and that names of interface methods are globally unique. Other features missing in CoreGI include imperative features, visibility modifiers, type erasure [Bracha et al. 1998], wildcards [Torgersen et al. 2004], inference of type arguments for method invocations [Gosling et al. 2005, § 15.12.2.7; Smith and Cartwright 2008], and interfaces as implementing types. The following discussion explains how the compiler for the full language handles these features. Other features of JavaGI not included in CoreGI are straightforward to implement.

5.1.1 *Imperative Features.* JavaGI does not introduce any new imperative features (with respect to Java) and most of Java's imperative features are orthogonal to the JavaGI-specific extensions. Thus, we conjecture that type soundness of CoreGI also holds in a setting with Java's imperative features. A minor problem arises when JavaGI's dynamic-dispatch algorithm for method invocations encounters **null** as one of the dispatch arguments. The implementation handles this case by throwing a NullPointerException, analogously to the case in Java where **null** appears as the receiver of a method invocation.

5.1.2 *Visibility Modifiers.* JavaGI fully respects Java's encapsulation properties. Inside implementation definitions, regular Java visibility rules apply; for example, **private** fields and methods of the implementing types are not accessible. JavaGI regards all implementation definitions as **public**.

5.1.3 *Type Erasure.* CoreGI's dynamic semantics is a type-passing semantics where type arguments are available at run time. In contrast, Java and the full JavaGI language perform type erasure during compilation, so type arguments are not available at run time.

The definition of CoreGI carefully avoids relying too much on run-time type arguments. For example, well-formedness criterion WF-IFACE-3 prevents implementing types from appearing nested inside argument types of method signatures and criterion WF-PROG-4 requires constraints of implementation definitions to be consistent with respect to the subtyping relationship among implementing types. Both criteria ensure that dynamic dispatch does not require run-time type arguments.

Elsewhere, the definition of CoreGI requires minor adjustments to work under a type-erasure semantics. For example, CoreGI's well-formedness criterion WF-PROG-1, which prevents overlapping implementation definitions, needs to be

adapted for the full language (see Section 3.4.1).

5.1.4 *Wildcards.* Proving type soundness for Java wildcards [Torgersen et al. 2004] is known to be a tricky business [Cameron et al. 2008]. Nevertheless, we believe that type soundness holds for the full JavaGI language including wildcards because JavaGI generalizes CoreGI's well-formedness criteria WF-IFACE-2 and WF-IFACE-3 to prevent implementing type variables such as **This** from appearing nested inside generic types at all. Thus, implementing type variables, which behave covariantly, never form upper or lower bounds of wildcards, the latter of which behave contravariantly.

Wildcards do not only challenge type soundness but also the decidability of subtyping. In general, it is still an open question whether subtyping for Java wildcards is decidable [Mazurak and Zdancewic 2006; Wehr and Thiemann 2009b]. Of course, the inclusion of wildcards in JavaGI is a concession to ensure backwards compatibility with Java 1.5. An embedding of JavaGI's generalized interface concept in a language like C# could easily drop support for wildcards. Thus, the decidability question for wildcards is not intrinsic to the decidability of subtyping in JavaGI.

5.1.5 *Inference of Type Arguments.* The JavaGI compiler supports inference of type arguments for method invocations by simply reusing Java's inference algorithm [Gosling et al. 2005, § 15.12.2.7]. Consequently, JavaGI-specific constraints in method signatures do not contribute to the improvement of type arguments. In general, this is not a problem because Java's inference algorithm is incomplete anyway [Smith and Cartwright 2008]. If inference fails, then the programmer may still invoke the method in question by specifying the type arguments explicitly.

JavaGI-specific features run no risk of introducing soundness-holes into the type inference process because the JavaGI compiler verifies correctness of inference during typechecking. This verification step is also needed for plain Java because Java's inference algorithm is unsound [Smith and Cartwright 2008].

5.1.6 *Interfaces as Implementing Types.* In Java, only classes may implement interfaces. Consequently, CoreGI supports only classes as implementing types of retroactive interface implementations. However, it is sometimes desirable to implement the methods of an interface in terms of the methods declared by some other interface. Section 2.3 already took advantage of the extension and defined a retroactive implementation of interface EQ with interface List<X> as the implementing type.

In its most general form, support for interfaces as implementing types renders subtyping—and hence typechecking—undecidable [Wehr and Thiemann 2008; 2009b; Wehr 2010]. However, we identified several reasonable restrictions that keep subtyping decidable and permit retroactive implementations such as the one of EQ for List<X> from Section 2.3 [Wehr and Thiemann 2009b]. The full JavaGI language supports interfaces as implementing types under one of these restrictions, as already explained in Section 3.4.5.

5.2 Translating JavaGI to Java Byte Code

The compilation scheme employed by the JavaGI compiler is based on a formal translation [Wehr 2010, Chapter 4] from a significant subset of CoreGI (see Sec-

```
// Java 1.4
import javagi.runtime.RT;
import javagi.runtime.Wrapper;
import java.util.*;
// 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$, Object that);
}
public class EQ$Wrapper extends Wrapper implements EQ {
  public static boolean eq$Dispatcher(Object this$, Object that) {
    Object dict = RT.getDict(EQ$Dict.class, EQ$Dict.eq$DispatchVector,
                             new Object[]{this$, that});
    return ((EQ$Dict) dict).eq(this$, that);
  }
  public EQ$Wrapper(Object obj) {
    super(obj); // The superclass constructors stores the wrapped object in a field.
  }
  public boolean eq(Object that) {
    // JavaGI compiler guarantees that this method is never called
    throw new Error("Binary method invoked on wrapper object");
  }
  // Superclass delegates hashCode, equals, and toString to the wrapped object.
}
// Translation of class Lists
class Lists {
  static int pos(Object x, List list) {
    int i = 0; Iterator iter = list.iterator();
    while (iter.hasNext()) {
      Object y = iter.next(); if (EQ$Wrapper.eq$Dispatcher(x, y)) return i; else i++;
    }
    return -1;
  }
}
```

Figure 25: Translation of interface EQ and class Lists from Section 2.2.

tion 4) to a slightly extended version of Featherweight Java [Igarashi et al. 2001]. It never modifies existing source code or byte code and supports retroactive interface implementations for arbitrary classes and interfaces, even if they are part of Java's standard library. Nevertheless, JavaGI supports in-place object adaptation; that is, new operations are available even for existing objects.

To demonstrate how the translation works, Figure 25 contains the translation of the interface EQ and the class Lists from Section 2.2, and Figure 26 contains the translation of the retroactive implementations defined in Section 2.2 and Section 2.3. For readability, the figures show Java 1.4 source code instead of the byte code generated by the JavaGI compiler.

5.2.1 *Translating Interfaces.* The JavaGI compiler generates a *dictionary interface* J$Dict for each interface J. For single-headed interfaces, it also generates a *wrapper class* J$Wrapper and a Java 1.4 interface J using Java's erasure translation [Igarashi et al. 2001; Gosling et al. 2005]. For example, erasure translates the type variable **This** of interface EQ to Object in the code in Figure 25.

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$Dict). Furthermore, the dictionary interface contains a *dispatch vector* of name m$DispatchVector for each non-static method m 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 multiple dispatch. For an n-headed interface, the dispatch vector is an int array of length $2n$ where, for $i \in \{0, \ldots, n-1\}$, the value at index $2i$ denotes the zero-based position of the implementing type corresponding to the receiver or argument whose position is stored at index $2i+1$. (Positions of argument types start at one, the receiver type has position zero.) For example, the receiver and the first argument of eq both refer to the implementing type **This** of EQ, so the dispatch vector in EQ$Dict is {0,0,0,1}.[19]

The wrapper class J$Wrapper serves as an adapter when a class is used at an interface type that it implements only retroactively. Most aspects of wrapper classes are standard [Baumgartner et al. 2002], but there are some JavaGI-specific issues. First, the eq method of EQ$Wrapper always throws an exception because JavaGI's type system ensures that such a binary method is never called on a wrapper object. (Section 3.1 explains why such a call would be unsound.)

Second, the wrapper class provides a static *dispatcher method* m$Dispatcher for every method m of the original interface. These dispatcher methods simplify the translation of retroactive method invocations. The dispatcher method for eq (named eq$Dispatcher) calls getDict from class javagi.runtime.RT, passing the class object for EQ's dictionary, the dispatch vector for eq, and an array containing the actual arguments. Based on this information, the run-time system returns a dictionary object corresponding to some retroactive implementation of EQ, through which the dispatcher invokes the eq method. For a non-binary method, the dispatcher would first try to invoke the method directly on this$, provided this$ implemented the method's declaring interface non-retroactively.

Section 5.3 explains how JavaGI's runtime system avoids the standard problems concerning wrappers, type identity, and object identity.

5.2.2 *Translating Invocations of Retroactively Implemented Methods.* 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 pos method of class Lists calls eq$Dispatcher defined in EQ$Wrapper (see Figure 25).

[19] It would be more natural to encode the dispatch vector as an n-element array of pairs of `int`s. However, Java does not support a primitive type for pairs, so we choose the alternative, flat representation.

```
// Java 1.4
import javagi.runtime.Dictionary;
import javagi.runtime.ImplementationInfo;
import java.util.*;
// Translations of EQ[Expr]
public class EQ$Dict$Expr implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    // load−time checks guarantee that this method is never called
    throw new Error("abstract method");
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
// Translation of EQ[IntLit]
public class EQ$Dict$IntLit implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    IntLit i1 = (IntLit) this$; IntLit i2 = (IntLit) that; return i1.value == i2.value;
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
// Translation of EQ[PlusExpr]
public class EQ$Dict$PlusExpr implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    PlusExpr e1 = (PlusExpr) this$; PlusExpr e2 = (PlusExpr) that;
    return EQ$Wrapper.eq$Dispatcher(e1.left, e2.left) &&
           EQ$Wrapper.eq$Dispatcher(e1.right, e2.right);
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
// Translation of EQ[List<X>]
public class EQ$Dict$List implements EQ$Dict, Dictionary {
  public boolean eq(Object this$, Object that) {
    List l1 = (List) this$; List l2 = (List) that;
    Iterator thisIt = l1.iterator(); Iterator thatIt = l2.iterator();
    while (thisIt.hasNext() && thatIt.hasNext()) {
      Object thisX = thisIt.next(); Object thatX = thatIt.next();
      if (! EQ$Wrapper.eq$Dispatcher(thisX, thatX)) return false;
    }
    return !(thisIt.hasNext() || thatIt.hasNext());
  }
  public ImplementationInfo _$JavaGI$implementationInfo() { ... }
}
```

Figure 26: Translation of retroactive implementations from Section 2.2 and Section 2.3.

5.2.3 *Translating Retroactive Interface Implementations.* Figure 26 presents the translation of the retroactive implementation definitions from Sections 2.2 and 2.3, again displaying Java 1.4 source code instead of byte code. 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$Dict$IntLit corresponds to the implementation

EQ [IntLit] and implements the dictionary interface EQ$Expr.

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 [IntLit] has type IntLit, 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$Dict$IntLit casts the arguments this$ and that from Object to IntLit, 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.

Besides the dictionary interface, each dictionary class also implements the interface javagi.runtime.Dictionary provided by JavaGI's runtime system. This interface requires a method _$JavaGI$implementationInfo that is used to reify information about the implementation. More specifically, the ImplementationInfo object returned by _$JavaGI$implementationInfo contains information about the type parameters, the interface, the implementing types, the constraints, and the abstract methods of the implementation. Further, it also specifies which of the implementing types are dispatch types.[20]

Figure 26 also contains the translation of the parameterized implementation of EQ for List<X> from Section 2.3. The resulting code demonstrates that the translation mechanism generalizes seamlessly to parameterized and type-conditional implementations. The translation of inheritance between implementation definitions (not shown in Figure 26) is also straightforward because it simply boils down to inheritance between the corresponding dictionary classes.

5.3 Run-Time System

JavaGI's run-time system maintains the available implementation definitions, checks their well-formedness according to the criteria in Section 3.4, loads new implementation definitions at run time, and performs dynamic dispatch on retroactively implemented methods. Moreover, it carries out certain cast operations, **instanceof** tests, and identity comparisons (==), for which the regular JVM instructions are not sufficient in the presence of wrappers [Baumgartner et al. 2002]. For example, to execute a JavaGI cast (J)obj, where J is an interface, the run-time system performs the following steps:

(1) Remove a potential wrapper around obj to arrive at object obj'.
(2) Check whether the run-time type T of obj' implements J.
(3a) If T implements J retroactively then the result of the cast is obj' wrapped by a J-wrapper.
(3b) If T implements J non-retroactively then the result of the cast is simply obj'.
(3c) If T does not implement J then the cast throws a ClassCastException.

[20] Section 3.4.2 and Figure 15 defined the notion of dispatch types.

The JavaGI-specific version of **instanceof** works similarly but evaluates to true in cases (3a) and (3b) and to false in case (3c). Performing an identity comparison x == y on two non-primitive values x and y requires the runtime system to remove potential wrappers around x and y (unless their static types are class types different from Object) before performing the corresponding JVM instruction.

JavaGI's implementation strategy for wrappers makes their presence almost transparent to the users of the language. There are only two possibilities to distinguish between a wrapped and an unwrapped object: use Java's reflection API, or inspect the object with a method compiled by a plain Java compiler.

Initialization of the run-time system happens lazily through a static initializer of class javagi.runtime.RT. The initializer code 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 described in Section 3.4. 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 and efficient method lookup.

Optionally, JavaGI's run-time system installs a custom class loader to monitor loading of classes and interfaces. The custom class loader assists in checking the well-formedness criteria described in Section 3.4.3 and Section 3.4.6. Without the custom class loader, the run-time system has to resort to conservative approximations of these criteria. The custom class loader could also automatically load the relevant retroactive implementations whenever java.lang.Class.forName(String name) is invoked, thus eliminating the need for the custom class loading method provided by JavaGI's runtime system.[21]

6. PRACTICAL EXPERIENCE

This section reports on practical experience with JavaGI and its implementation. More specifically, it describes three real-world case studies performed with JavaGI and it presents benchmark data showing that the code generated by the JavaGI compiler offers good performance. The source code of the case studies and the benchmarks is available on the JavaGI homepage [Wehr 2009].

6.1 Case Studies

We report on three real-world case studies to examine the benefits of generalized interfaces and to demonstrate the practical utility of the JavaGI implementation. Here, we only give a summary of each case study, more details can be found elsewhere [Wehr and Thiemann 2009a].

6.1.1 *XPath Evaluation.* This case study deals with a framework for evaluating XPath [Clark and DeRose 1999] 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, the jaxen project [Jaxen 2008] already provides such a framework. The goal of the case study was to provide a JavaGI solution and compare it to the Java solution provided by

[21] The current implementation does not support this feature, though.

jaxen.

The JavaGI solution specifies a model of the XPath node hierarchy based on interfaces. These interfaces contain the methods required to perform the actual evaluation of XPath expressions. A JavaGI programmer then adapts existing object models to the XPath node hierarchy by using retroactive interface implementations.

The case study adapts the two well-known Java XML libraries dom4j [dom4j 2008] and JDOM [Hunter and McLaughlin 2007] to the XPath node hierarchy. Dom4j and JDOM are sufficiently different because dom4j represents XML nodes using an interface hierarchy, whereas JDOM relies on non-hierarchically organized classes for this purpose. The case study shows that both libraries can be adapted to the XPath node hierarchy elegantly and without code duplication. Compared with the plain Java solution provided by jaxen, the JavaGI approach requires significantly fewer casts: with JavaGI, no casts at all are needed to adapt dom4j and JDOM to the XPath node hierarchy, whereas the adaptations of these two libraries to jaxen's XPath evaluation interface requires 28 and 47 casts, respectively. Section 6.2 compares the JavaGI and the Java solution with respect to performance.

6.1.2 JavaGI *for the Web.* WASH [Thiemann 2005] is a domain specific language for server-side Web scripting embedded in Haskell [Peyton Jones 2003]. It supports the generation of HTML in a way that guarantees 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.

The implementation of WASH relies heavily on Haskell's type classes. It enforces (not quite) the validity of generated HTML by providing type classes specifying the permitted 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.

The case study implements much of WASH's core functionality in JavaGI. The resulting application runs inside Tomcat [Apache Software Foundation 2009], an implementation of the Java Servlet specification [Sun Microsystems 2007]. Briefly put, plain Java interfaces are sufficient to support generation of (not quite) valid HTML documents, retroactive implementation is useful in many places, the implementation of typed input widgets benefits from static interface methods, and dynamic loading of implementations is essential for working in a servlet environment.

6.1.3 *Java Collection Framework.* The Java Collection Framework (JCF [Sun Microsystems 2004a]) provides interfaces for data structures such as Collection, Set, List, and Map as well as various implementations of these data structures. By default, all collections are modifiable but programmers can explicitly mark a collection as unmodifiable. However, unmodifiable collections have the same interface as modifiable ones, so a programmer may inadvertently call modifying operations on unmodifiable collections, resulting in run-time errors.

Huang, Zook, and Smaragdakis [2007] demonstrate how to turn such run-time errors into compile-time errors using type conditionals as provided by their Java ex-

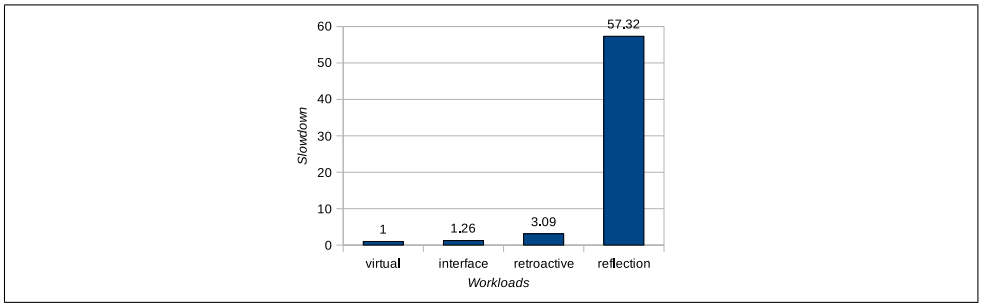

Figure 27: Micro benchmarks for different kinds of method call instructions.

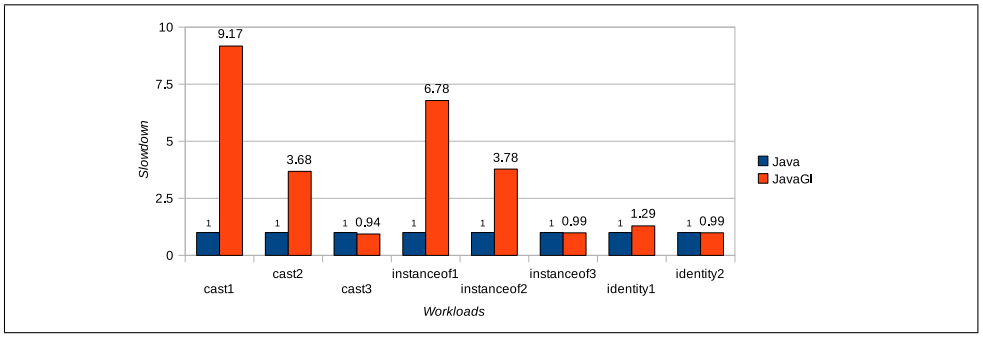

Figure 28: Micro benchmarks for cast operations, **instanceof** tests, and identity comparisons.

tension cJ. The basic idea is to parameterize a collection not only over the element type but also over a "mode" type that specifies further attributes of a collection. Type conditionals then ensure that operations modifying a collection are only available if the mode parameter indicates that the collection is indeed modifiable.

Our case study performs a similar refactoring of the JCF using JavaGI's type conditionals. The main difference between the JavaGI and the cJ versions of the refactoring is that cJ offers a grouping mechanism for type conditionals. This grouping mechanism allows a programmer to specify a type conditional for a whole group of methods, whereas JavaGI requires restating the condition for each method.

Furthermore, in cJ superclasses and fields are also subject to type conditionals. However, these features are not needed for the JCF case study, and the original cJ paper [Huang et al. 2007] does not contain realistic examples using them.

6.2 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 good 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 Virtual Machine (JVM [Lindholm and Yellin 1999]) used was the server virtual machine of Sun's Java SE (version 1.6.0_06 [Sun Microsystems 2009]). The Java code was compiled

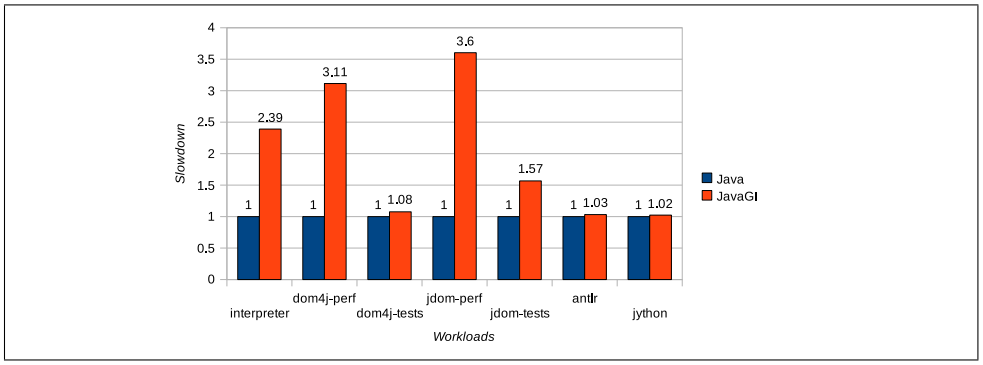

Figure 29: Performance of JavaGI with respect to Java.

with the Eclipse Compiler for Java (version 0.883_R34x [Eclipse Foundation 2008]), the baseline of the JavaGI compiler. The JavaGI code was compiled with the JavaGI compiler presented in Section 5.

The individual workloads were repeatedly executed, until performance stabilized. The mean of the last three or five repetitions (depending on the total number of repetitions) then represents the performance index for a workload. The raw benchmark data is available online [Wehr 2009].

Figure 27 shows performance results of micro benchmarks demonstrating 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 calls using the `invokeinterface` instruction. This slowdown is not surprising because the machinery needed to perform dynamic lookup of retroactively implemented methods is more involved than that required for ordinary class or interface methods (see Section 5.2.2). For reference, the figure also includes the slowdown of method calls via reflection.

Figure 28 compares cast operations, **instanceof** tests, and identity comparisons (==) in Java and JavaGI. The workload *cast1* casts objects to an interface that these objects implement directly in the Java version but retroactively in the JavaGI version. In general, casts are complex operations in JavaGI (see Section 5.3), so the JavaGI version is 9.17 times slower than the Java version.[22] The workload *cast2* casts objects with static type Object to a class type. The JavaGI version is 3.68 slower than its Java counterpart because such casts require unpacking (but no re-packing as for cast1) of potential wrappers. The workload *cast3* casts objects whose static types are class types other than Object to some other class type. In such situations, the JavaGI compiler generates a regular `checkcast` JVM instruction, so there is no significant difference between the Java and the JavaGI versions. The workloads *instanceof1*, *instanceof2*, and *instanceof3* are similar to cast1, cast2, and cast3, respectively, but perform **instanceof** tests instead of cast operations. The

[22] Under a different workload [Wehr and Thiemann 2009a], casts in JavaGI were up to 830 times slower than in Java. However, this different workload is unrealistic because it performs repeated cast operations always on the *same* object. In this scenario, a caching mechanism of the JVM apparently leads to very fast execution times. In contrast, workload cast1 is more realistic because it uses *different* objects to measure to performance of cast operations.

workload *identity1* checks whether two objects with static type Object are identical using the == operator. The JavaGI version is slower because it involves unpacking of potential wrappers. The workload *identity2* is similar but compares objects whose static types are class types different from Object. In this case, no unpacking is required, so there is no significant performance difference.

Figure 29 compares the performance of JavaGI with that of Java using seven real-world workloads. The *interpreter* workload is an interpreter for a language with arithmetic expressions, variables, conditionals, and function calls, implemented once in plain Java and once in JavaGI. The Java interpreter uses ordinary virtual methods to perform expression evaluation, whereas the JavaGI interpreter uses retroactively implemented methods for this purpose. The JavaGI version is 2.39 times slower than the Java version. The large number of calls to retroactively implemented methods in the JavaGI version causes this slowdown.

The workloads *dom4j-perf*, *dom4j-tests*, *jdom-perf*, and *jdom-tests* are from the jaxen [2008] distribution (see Section 6.1.1). Dom4j-perf and jdom-perf are performance tests for the adaptation of dom4j [2008] and JDOM [Hunter and McLaughlin 2007] to jaxen's XPath evaluation framework, dom4j-tests and jdom-tests are the corresponding unit-test suites. The Java versions of these workloads directly use the code from the jaxen distribution, whereas the JavaGI versions replace jaxen's XPath evaluation framework with the framework presented in Section 6.1.1.

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 3.11 and 3.6, respectively, times slower than their Java counterparts.[23] 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 slowdown. (Many 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.)

The workloads *antlr* and *jython* in Figure 29 are from the DaCapo benchmark suite (version 2006-10-MR2 [Blackburn et al. 2006]). 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.

7. RELATED WORK

An article [Wehr et al. 2007] at ECOOP 2007 presented the initial design of JavaGI. The language introduced in that article included full-blown bounded existential types and omitted many restrictions, thus rendering subtyping and typechecking undecidable. The undecidability results were established in two papers [Wehr and

[23] A slight variation of the workloads dom4j-perf and jdom-perf [Wehr and Thiemann 2009a] increases the performance of the Java versions. In this setting, the workloads for JavaGI are 3.59 and 5.73, respectively, times slower than their Java counterparts. The variation, however, is quite unrealistic because it evaluates an XPath expression repeatedly on the *same* object graph. In contrast, the original dom4j-perf and jdom-perf workloads are more realistic because they use *different* object graphs to evaluate XPath expressions on.

Thiemann 2008; 2009b] at FTfJP 2008 and APLAS 2009. Apart from decidability issues, the ECOOP paper did not define a dynamic semantics, so there was no implementation and the type soundness proof was not worked out. Furthermore, the translation scheme sketched in the ECOOP paper was too limiting because it did not support dynamic loading of implementation definitions and required severe restrictions on their locations (see Section 7.5). In contrast, JavaGI as presented in this article 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. It comes with a formalization that enjoys type soundness, decidable subtyping and typechecking, as well as deterministic evaluation. A recent paper at GPCE 2009 [Wehr and Thiemann 2009a] presented the implementation of a compiler and a run-time system for JavaGI (Section 5), as well as case studies and benchmark results (Section 6).

7.1 Type Classes in Haskell

Type classes [Wadler and Blott 1989; Kaes 1988; Jones 1994] in the functional programming language Haskell [Peyton Jones 2003] are closely related to generalized interfaces. Like a generalized interface, a type class declares the signatures of its member functions depending on one or more implementing types.[24] Unlike in JavaGI, however, member functions of type classes are not attached to some receiver object but denote top-level functions that may be overloaded for different types. Thus, methods of Haskell type classes are similar to static interface methods in JavaGI. One difference is that Haskell infers, at least in most cases, the instance from which a method should be taken, whereas this information has to be specified explicitly in JavaGI (for static interface methods, that is). Haskell's type classes support multiple inheritance, just as interfaces in JavaGI do. Further, both languages provide constraint mechanisms to limit possible instantiations of universally quantified type variables. In contrast to JavaGI, Haskell infers constraints and types automatically. Like JavaGI's retroactive interface implementations, Haskell's instance definitions specify that one or several types are members of some type class, thereby providing overloaded versions of the member functions of the class. As in JavaGI, instances are defined in separation from types; they can be parametric and subject to constraints.

Functional dependencies [Jones 2000], a well-known extension of Haskell type classes, express dependencies between implementing types. For example, given the declaration **class** C a b | a → b ... of a two-parameter type class C, the functional dependency a → b specifies that in all instances of C the first implementing type uniquely determines the second. Such dependencies are to some degree expressible in JavaGI because its type system (as well as Java's) requires that a program does not define two implementations for different instantiations of the same interface (see Section 3.4.2 and criterion WF-PROG-2 in Section 4.4.3). For example, the JavaGI interface **interface** I<b>[a] ... specifies the same dependency between the type variables a and b as the type class C just presented. More complex functional dependencies such as a → b, b → a are not expressible in JavaGI. Associated

[24]The Haskell 98 standard [Peyton Jones 2003] supports only one implementing type, but the commonly implemented multi-parameter type classes [Peyton Jones et al. 1997] lift this restriction.

types [Myers 1996; Chakravarty et al. 2005; Chakravarty et al. 2005] present an alternative to functional dependencies (see Section 7.2).

Haskell generalizes type classes to constructor classes [Jones 1993], where the implementing types are in fact type constructors. JavaGI only supports first-order parametric polymorphism (as Java does). We conjecture that lifting this restriction [Moors et al. 2008] would permit a mechanism similar to constructor classes for JavaGI. Definitions of type classes in Haskell may provide default implementations for the methods of the type class. JavaGI can encode such default implementations with abstract implementations and implementation inheritance (see Section 2.5).

In comparison with object-oriented languages, Haskell has neither subtyping nor dynamic dispatch. Thus, Haskell can construct evidence for type-class instances needed in a function body statically or from the evidence present at the call sites of the function. This approach is too limiting for JavaGI because it either prevents dynamic dispatch or severely restricts the choice of compilation units into which retroactive implementations can be placed. Hence, one major contribution of JavaGI with respect to Haskell is the type-safe integration of subtyping and dynamic dispatch with type classes. Another difference between the two languages is that type classes only constrain types but never appear as types on their own. (There exists, however, an extension [Thiemann and Wehr 2008] that provides exactly this feature.) In contrast, JavaGI's single-headed interfaces can be used in constraints and as types.

The OOHaskell project [Kiselyov and Lämmel 2005] shows how Haskell 98 with common extensions supports many object-oriented programming idioms such as encapsulation, mutable state, inheritance, and overriding. Essentially, OOHaskell builds on extensible polymorphic records from the HList library [Kiselyov et al. 2004] and on a semi-explicit subsumption operation. The approaches of OOHaskell and JavaGI are different: OOHaskell emulates object-oriented programming in Haskell, whereas JavaGI extends an object-oriented programming language with features influenced by Haskell.

7.2 Generic Programming

Concepts for C++ borrow ideas from Haskell type classes to specify requirements on template parameters [Myers 1996; McNamara and Smaragdakis 2000; Järvi et al. 2003; Reis and Stroustrup 2006; Gregor et al. 2006; Bernardy et al. 2008]. The main motivation behind concepts is to improve error messages caused by malformed template instantiations and to enable separate compilation for templates. Like type classes and generalized interfaces, concepts may span multiple types, they support some form of inheritance, and they can appear in constraints. In addition, concepts can also contain type definitions, leading to the notion of associated types [Myers 1996]. There are two choices for implementing a concept with existing types [Gregor et al. 2006]: either programmers provide explicit concept maps (similar to retroactive interface implementations in JavaGI and instance definitions in Haskell) or the compiler derives an implicit implementation based on the types and operations in scope. Like retroactive implementations, concept maps can be parametric and subject to type conditions. Siek and Lumsdaine [2005] provided a formalization of concepts as an extension to System F [Girard 1972; Reynolds 1974]. The first author extended this formalization to a realistic programming

language $\mathcal{G}$ [Siek 2005], which he then used to provide prototype implementations of the Standard Template Library [Stepanov and Lee 1995] and the Boost Graph Library [Siek et al. 2002]. The main difference between concepts and JavaGI's generalized interfaces is that concepts are resolved at compile time: the compiler instantiates a template parameter based on the most specific implementations of the concepts imposed on the parameter. Consequently, languages such as $\mathcal{G}$ make use of a compilation scheme similar to the dictionary translation popularized for Haskell type classes [Hall et al. 1996] (see also Section 7.1). In contrast, JavaGI resolves methods of retroactive implementations dynamically through multiple dispatch.

Another difference between concepts and JavaGI's retroactive implementations is that concept maps are lexically scoped whereas retroactive implementations share a global scope. Further, the concept mechanism as presented by Gregor et al. [2006] supports concept-based overloading, same-type and negative constraints, and constraint propagation [Järvi et al. 2005]. The idea of negative constraints conflicts with JavaGI's open-world assumption for retroactive implementations. Concept-based overloading is not available in JavaGI because neither static nor dynamic resolution of overloading based purely on concepts (i.e. implementation constraints) is possible due to JavaGI's open-world assumption and its type-erasure semantics, respectively.

A comparative study [Garcia et al. 2003; 2007] identified eight features that are important to properly support generic programming. Apart from associated types and the closely related feature of type aliases, JavaGI supports all of them, including two properties ("multi-type concepts" and "retroactive modeling") not supported by Java. Other researchers proposed associated types as extensions of Haskell type classes [Chakravarty et al. 2005; Chakravarty et al. 2005] and C# [Järvi et al. 2005], so we conjecture that their addition to JavaGI does not pose significant challenges. The work of Järvi and colleagues also includes constraint propagation and same-type constraints, indicating that these feature could be added to JavaGI without major difficulties.

7.3 Family Polymorphism

Traditional polymorphism fails to express collaborations between families of types in a way that is both type safe (mixing objects from different families is rejected at compile time) and generic (abstraction over the family per se is possible). Ernst [2001] suggested family polymorphism as a solution to this problem. His running example demonstrates how virtual patterns (a form of virtual classes [Madsen and Møller-Pedersen 1989]) in gbeta [Ernst 1999] express a type-safe and generic abstraction over graphs, where a graph is seen as a collaboration of two family members "node" and "edge". JavaGI can encode Ernst's graph example using multi-header interfaces [Wehr 2010, Figure 8.3] (see also Section 2.7). The main difference between the encoding based on multi-headed interfaces and Ernst's solution with virtual patterns is that the former represents families at the type level, whereas the latter represents them on the value level. A type-level representation has several disadvantages compared with a value-level representation: only a fixed number of distinct families can be defined and only classes not related by subclassing can form distinct families (i.e., if classes $\mathsf{C}_1, \ldots, \mathsf{C}_n$ belong to some family then $\mathsf{C}'_1, \ldots, \mathsf{C}'_n$ usually belong to the same family in JavaGI if each C'_i is a subclass of C_i).

A drawback of the value-level representation is that it complicates the type system a lot. Ernst's solution permits the construction of heterogeneous data structures over families. In general, such data structures are possible in JavaGI but their encoding is quite complex and hardly usable in practice (it relies on the well-known trick to simulate existential types through continuations and rank-2 polymorphism).

Other approaches to family polymorphism include Scala's abstract type members with self-type annotations [Odersky and Zenger 2005b], OCaml's object system [Leroy 2008; Rémy and Vouillon 1997; Rémy and Vouillon 1998a; 1998b], variant path types [Igarashi and Viroli 2007], lightweight family polymorphism in the context of Java [Saito et al. 2008], type parameter members [Kamina and Tamai 2007], lightweight dependent classes [Kamina and Tamai 2008], Helm and coworkers' contracts [1990], and a generalization of `MyType` [Bruce et al. 1995; Bruce et al. 2003] to mutually recursive types [Bruce et al. 1998]. The last approach bears close resemblance to JavaGI's multi-headed interfaces but relies on exact types to prevent unsoundness in the presence of binary methods, whereas JavaGI uses multiple dispatch instead. (Section 7.6 discusses `MyType` and exact types in more detail.)

7.4 Software Extension, Adaptation, and Integration

Many research projects address better support for software extension, adaptation, and integration. This section discusses work most relevant to JavaGI.

7.4.1 *The Expression Problem.* The expression problem, which goes back to Reynolds [1975; 1994] and Cook [1991] but was popularized under its name by Wadler [1998], highlights a key problem in the area of software extensibility: how to extend a given data structure modularly in the dimensions of data *and* operation. Torgersen [2004] defined a solution to the expression problem as a "combination of a programming language, an implementation of a Composite structure in that language, and a discipline for extension which permits both new data types and operations to be subsequently added any number of times, without modification of existing source code, without replication of non-trivial code, without risk of unhandled combinations of data and operations." JavaGI's approach to the expression problem, as outlined in Section 2.1, fulfills these requirements. Torgersen also evaluated solutions to the expression problem according to their degree of extensibility: "code-level extensibility" requires that existing code can be extended without recompilation, and "object-level extensibility" requires that objects created before introducing an extension remain valid and compatible afterwards. JavaGI provides both kinds of extensibility. An additional requirement [Odersky and Zenger 2005a] is that a solution to the expression problem must typecheck statically and that it must be possible to combine independently developed extensions. JavaGI fulfills both of these requirements (assuming that the independently developed extensions are sufficiently disjoint), although typechecking in JavaGI is not fully modular.

7.4.2 *Solutions with Type Classes in Haskell.* Lämmel and Ostermann [2006] showed how Haskell type classes solve several extensibility, adaptability, and integration problems that have been used to illustrate limitations of object-oriented languages. Their Haskell solutions to the adapter problem [Gamma et al. 1995], the tyranny of the dominant decomposition problem [Harrison and Ossher 1993; Ossher and Tarr 1999], the expression problem [Wadler 1998; Torgersen 2004], the

framework integration problem [Mattsson et al. 1999; Mezini et al. 2000], and the problem of providing family polymorphism Ernst [2001] can be ported to JavaGI easily. However, their approach to multiple dispatch differs from that in JavaGI because Haskell does not support dynamic dispatch as already discussed in Section 7.1 (see also Section 7.5).

7.4.3 *Virtual and Nested Classes.* Section 7.3 discussed virtual classes [Madsen and Møller-Pedersen 1989] in the context of family polymorphism [Ernst 2001]. But virtual classes also enable solutions to a number of extensibility problems. Higher-order hierarchies [Ernst 2003] allow programmers to extend, combine, and modify existing class hierarchies. The main features enabling this kind of extensibility are furtherbinding (virtual classes are not overridden but enhanced in subclasses) and virtual superclasses (superclass declarations are subject to late binding). JavaGI's retroactive interface implementations also enable the extension of existing class hierarchies with new functionality. Although changing existing hierarchies is not possible in JavaGI, retroactive interface implementations enable the introduction of new superinterfaces for existing classes and interfaces. The combination of extensions is implicit in JavaGI because retroactive interface implementations perform in-place object adaptation, whereas higher-order hierarchies create new copies of existing hierarchies and thus need an explicit combine operator. This copy-based approach prevents extensions from being available for existing class instances, a limitation not shared by JavaGI. Further, adding functionality to existing classes in the style of higher-order hierarchies seems to require a default implementation for the root of the class hierarchy, whereas JavaGI avoids the need for such default implementations by permitting abstract methods in retroactive implementations. Completeness checking for abstract methods requires load-time checks, though. Higher-order hierarchies support the addition of state (i.e., instance variables) to existing classes but JavaGI does not.

Nested inheritance [Nystrom et al. 2004] also supports the extension of class hierarchies through nesting and furtherbinding of classes. Unlike virtual classes, nested inheritance treats a nested class as an attribute of its enclosing class. Nested intersection [Nystrom et al. 2006] generalizes nested inheritance and enables the composition of class hierarchies by some form of multiple inheritance. As higher-order hierarchies, nested inheritance and nested intersection both follow a copy-based approach and make extensions not available for instances of existing classes. Class sharing [Qi and Myers 2009] adds support for in-place object adaptation to nested intersection: a sharing relation between classes implies that shared classes have the same set of object instances. Each shared class is a distinct view of such an instance, and an explicit operation may change that view. JavaGI does not require an explicit operation to combine different extensions. The extension mechanisms of JavaGI and nested inheritance are quite different: the former uses retroactive interface implementations, the latter inheritance. None of these mechanisms is superior to the other. From a programmers point of view, the additional complexity introduced by JavaGI seems to be lower than that of nested inheritance and its successors: JavaGI's additional features are all driven by a generalization of interfaces, whereas nested inheritance/intersection and class sharing confronts the programmer not only with a generalization of inheritance but also with a complex

type language making use of exact types, view-dependent types, prefix types, mask types, and sharing constraints [Qi and Myers 2009].

Collaboration interfaces [Mezini and Ostermann 2002] permit the declaration of types for components as a set of mutually recursive types by treating a nested interface as virtual. Moreover, collaboration interfaces provide support for expressing not only the provided but also the required services of a component. While JavaGI addresses the problem of specifying mutually recursive types through multi-headed interfaces, there is no support for expressing required services. Conversely, collaboration interfaces do not take retroactive implementation into account, so it might be worthwhile to investigate how a combination of collaboration interfaces and JavaGI's generalized interfaces would look like. The work on collaboration interfaces also suggested wrapper recycling to avoid object schizophrenia [Sekharaiah and Ram 2002; Hölzle 1993] caused by wrappers. Essentially, wrapper recycling ensures that there exists at most one wrapper for each interface/object combination, thus avoiding object schizophrenia between two wrapped objects but not between a wrapped and an unwrapped object. JavaGI deals with object schizophrenia by using special cast operations, **instanceof** tests, and identity comparisons (==). This approach avoids object schizophrenia also between wrapped and unwrapped objects but does not work as soon as a wrapped object is passed to a method that has not been compiled with the JavaGI compiler.

Virtual classes express dependencies between classes and objects through nesting. Hence, a class may depend at most on one object. Dependent classes [Gasiunas et al. 2007] replace nesting by parametrization and so permit dependencies between a class and multiple objects. Further, the type parameters of a class are subject to dynamic dispatch, so dependent classes complement multimethods (see Section 7.5) by providing multi-dispatched abstractions.

7.4.4 *Advanced Separation of Concerns.* Subject-oriented programming [Harrison and Ossher 1993; Ossher and Tarr 1999] and work on multi-dimensional separation of concerns [Tarr et al. 1999] deals with the tyranny of the dominant decomposition problem. This problem arises because most languages support only one fixed decomposition of a system, even though other decompositions might be meaningful and appropriate. Hyper/J [Ossher and Tarr 2000] provides multi-dimensional separation of concerns for Java. The tool allows an existing application to be decomposed into hyperslices, and it offers the possibility to define new hyperslices from scratch. Hyperslices represent different decompositions of a system and allow developers to view a system from different perspectives. A flexible composition mechanism then creates a full Java class from several hyperslices. As mentioned in Section 7.4.2, Lämmel and Ostermann use Haskell type classes to emulate some of the functionality of hyperslices [2006, Section 2.3]. Their solution also works in JavaGI, so Lämmel and Ostermann's comparison with Hyper/J remains valid in the context of the JavaGI language.

Aspect-oriented programming [Kiczales et al. 1997] is another technique for improving separation of concerns. It allows programmers to express crosscutting concerns (called aspects) in an explicit and modular manner. AspectJ [Kiczales et al. 2001; AspectJ Team 2009b; 2009a], an aspect-oriented extension of Java, provides two kinds of crosscutting concerns: dynamic crosscutting supports the definition of

additional code to run at certain points in the execution of a program; and static crosscutting affects the static type signature of a program. Inter-type member declarations and the `declare parents` form, two examples for static crosscutting, offer functionality similar to JavaGI's retroactive interface implementations: the former enable the addition of new members to existing classes, whereas the latter permits changes to the inheritance structure of a program by inserting new superinterfaces. There is no notion of dynamic crosscutting in JavaGI. The current implementation of AspectJ relies on compile-time weaving to support inter-type member declarations and the `declare parents` form [AspectJ Team 2009a, Chapter 5]. That is, the AspectJ compiler rewrites the byte code of the relevant classes, so it is not possible to modify classes that are not under the control of the compiler (e.g., classes from Java's standard library). In contrast, JavaGI never changes the byte code of existing classes, so it is possible to update arbitrary classes. Moreover, AspectJ's invasive compilation strategy causes changes to be visible to all clients of a class, whereas JavaGI guarantees that modifications do not change the behavior of existing clients.

7.4.5 *Module Systems for Java.* Keris [Zenger 2005] adds a module system to Java that permits type-safe addition, refinement, replacement, and specialization of modules without pre-planning. The resulting language provides composition of modules through nesting and infers module dependencies automatically. As in JavaGI, Keris' extensibility mechanism does not require source-code access and preserves the original version of a module being extended. The main difference between Keris and JavaGI is that the former introduces a new language construct (modules) whereas the latter makes an existing construct (interfaces) more powerful.

Inspired by MzScheme's [2009] units [Flatt and Felleisen 1998], Jiazzi [McDirmid et al. 2001] also enhances Java with module-like constructs to provide better support for component-based development. Unlike JavaGI and Keris, however, Jiazzi does not directly extend the Java language but introduces an external language for specifying package signatures and for defining and linking units. The system supports separate compilation, cyclic linking, and mixins [Bracha and Cook 1990], and it permits the modular addition of new features to existing classes. In contrast to JavaGI, Jiazzi requires all extensions of a class to be integrated into one module. Further, Jiazzi does not support dynamic loading of extensions.

Other work on module systems for Java include JavaMod [Ancona and Zucca 2001], JAM [Strniša et al. 2007; Sun Microsystems 2006], and Component NextGen [Sasitorn and Cartwright 2007].

7.4.6 *Statically Scoped Extension Mechanisms.* Classboxes [Bergel et al. 2003; Bergel et al. 2005; Bergel et al. 2005] offer scoped refinement of classes. Refining a class either adds a new feature (i.e., method, field, superinterface, constructor, inner class) or redefines an existing one, thereby creating a new version of the class. A scoping mechanism ensures that refinements are only locally visible so that potentially conflicting refinements can coexist inside the same program. In contrast, JavaGI's retroactive interface implementations can only add new methods and superinterfaces to classes, additions of other features and redefinitions are not possible. Further, retroactive interface implementations in JavaGI share a global

scope so two implementations of the same interface for the same class lead to a conflict. On the other hand, the compilation strategy for classboxes in Java [Bergel et al. 2005] is not modular because the compiler weaves all refinements of a class into the declaration of the class. The JavaGI compiler, however, supports non-invasive and modular code generation. Furthermore, classboxes do not provide multiple dispatch and advanced typing constructs such as explicit implementing types, multi-headed interfaces, and type conditionals.

Expanders [Warth et al. 2006] are quite similar to classboxes. They offer statically scoped, retroactive extension of classes with new fields, methods, and superinterfaces. The work on expanders also emphasizes modularity: a class may have multiple, independent extensions at the same time, but in each scope only explicitly opened extensions are visible. Unlike classboxes, however, expanders offer modular typechecking and compilation. JavaGI only offers mostly modular typechecking and fully modular compilation. Different from JavaGI, expanders impose some restrictions on the placement of extension code. For example, consider a class hierarchy contained in compilation unit $\mathcal{U}_1$, an extension of the class hierarchy (either through expanders or through retroactive interface implementations) in $\mathcal{U}_2$, and a refinement of the class hierarchy by standard subclassing in $\mathcal{U}_3$. Now suppose that the extension in $\mathcal{U}_2$ should be augmented to take the refinement in $\mathcal{U}_3$ into account. With expanders their are two options, neither of which is satisfactory: either edit the code in $\mathcal{U}_2$ to make the augmentation globally effective or provide a locally overriding expander in some compilation unit $\mathcal{U}_4$ to make the augmentation only visible from within $\mathcal{U}_4$. In contrast, JavaGI's retroactive implementation definitions enable a globally effective augmentation without touching the code in $\mathcal{U}_2$. Moreover, expanders do not support abstract methods, which may result in unwanted run-time exceptions because a reasonable default implementation of an operation does not always exist [Millstein et al. 2003]. Last not last, expanders provide neither multiple dispatch nor JavaGI's advanced typing constructs.

Concepts in C++ and Siek's language $\mathcal{G}$ [Siek 2005] also provide statically scoped extension mechanisms. We already discussed these two mechanisms in Section 7.2.

7.4.7 *Miscellaneous.* Hölzle [1993] argued that minor incompatibilities between independently developed components are unavoidable. Further, he discussed several mechanisms for dealing with such incompatibilities. JavaGI's retroactive interface implementation is a realization of his type adaptation proposal. Binary component adaptation [Keller and Hölzle 1998] supports the adaptation and evolution of components in binary form by rewriting component binaries at load-time. In contrast, JavaGI never changes the byte code of existing classes.

Scala [Odersky 2009] supports implicit parameters and methods, which can be used to define implicit conversions called views. A view from type `T` to interface `I` may simulate a retroactive implementation of `I` for `T`. However, unlike JavaGI's multiple dynamic dispatch, view selection in Scala is based on a single static type. Further, the implementation of a view often uses explicit wrappers, which suffer from object schizophrenia [Sekharaiah and Ram 2002; Hölzle 1993].

Partial classes in C# 2.0 [ECMA International 2002] provide a primitive, code-level modularization tool. The different partial slices of a class (comprising superinterfaces, fields, methods, and other members) are merged by a preprocessing phase

of the compiler. Extension methods in C# 3.0 [ECMA International 2005] support full separate compilation, but the added methods cannot be virtual, and members other than methods cannot be added.

Smalltalk [Goldberg and Robson 1989] and Objective-C [Apple Inc 2009] support the extension of existing classes with new methods. However, Smalltalk is dynamically typed, and Objective-C supports retroactive interface implementations only by resorting to the dynamically-typed part of the language.

7.5 External Methods and Multiple Dispatch

This section complements the preceding one by discussing work on external methods in combination with multiple dispatch. External methods permit extensions of existing classes by defining methods outside of class definitions. Their common motivation is to supersede the Visitor and the Adapter design patterns [Gamma et al. 1995] and to solve the expression problem [Wadler 1998; Torgersen 2004] (see Section 7.4.1). Multiple dispatch is the ability to perform dynamic dispatch not only on the receiver but also on the other arguments of a method call. This generalization of the traditional object-oriented dispatch mechanism subsumes binary methods [Bruce et al. 1995] (see also Section 7.6) and improves code modularity and readability by avoiding double dispatch [Ingalls 1986] and cascaded **instanceof** tests.

The combination of external methods and multiple dispatch is found in languages such as Common Lisp [Steele 1990], Dylan [Shalit 1997], Cecil [Chambers and Leavens 1996; Chambers 1992; Chambers and the Cecil Group 2004], as well as in the Java extension MultiJava [Clifton et al. 2000; Clifton et al. 2006] and its successor Relaxed MultiJava [Millstein et al. 2003]. JavaGI supports multiple dispatch through multi-headed interfaces and explicit implementing types: a method m that should dispatch on the receiver and on the arguments at positions $i_1, \ldots, i_{n-1}$ is encoded as an n-headed interfaces with implementing types $X_0, \ldots, X_{n-1}$ such that receiver X_0 defines method m and, for $k \in \{1, \ldots, n-1\}$, X_k is the i_kth argument type of m (see [Wehr 2010, Figure 8.4] for a concrete example). Declaring the signature of a multimethod in an interface fixes the dispatch positions for all implementations of the method in advance. The language Tuple [Leavens and Millstein 1998] shares this restriction with JavaGI, whereas Common Lisp, Dylan, Cecil, and (Relaxed) MultiJava permit different dispatch positions for different implementations.

The main problem in fitting multiple dispatch to a typed object-oriented language is modular typechecking without imposing too many restrictions. Common Lisp and Dylan are both dynamically typed, so the problem does not occur in these languages. Cecil requires the whole program to perform typechecking. The core language Dubious [Millstein and Chambers 1999] investigates what restrictions are necessary to support modular typechecking. The outcome of this investigation are three different systems: System M supports fully modular typechecking at the price of losing expressiveness; System E maximizes expressiveness but requires some regional typechecking and a simple link-time check; System ME lets programmers decide on a case-by-case basis whether to use System M or System E. All three systems are type sound, viz., neither "message-not-understood" nor "message-ambiguous" errors can occur at run time. The core calculus of JavaGI defined in Section 4 also

enjoys type soundness in this sense.

MultiJava's design [Clifton et al. 2000; Clifton et al. 2006] is based on System M. It supports fully modular typechecking at the price of several restrictions. JavaGI's initial design [Wehr et al. 2007] followed MultiJava and reformulated the restrictions as follows: (i) retroactively defined methods must not be abstract; and (ii) if an implementation of interface I in compilation unit $\mathcal{U}$ retroactively adds a method to class C, then $\mathcal{U}$ must contain either C's definition or an implementation of I for a superclass of C. These two restrictions permit modular typechecking but also severely limit expressiveness. The current version of JavaGI takes the same approach as Relaxed MultiJava [Millstein et al. 2003] and defers some checks to link time. These link-time checks enable JavaGI to drop the two restrictions just mentioned. JavaGI's well-formedness criterion "downward closed" (see Section 3.4.3) is directly influenced by Relaxed MultiJava's unambiguity check [Millstein et al. 2003, Section 3.3.4]. Both Relaxed MultiJava and JavaGI support fully modular code generation.

Dylan, Cecil, (Relaxed) MultiJava, and JavaGI all rely on symmetric multiple dispatch: They treat all dispatch arguments identically. Only few approaches (e.g., Common Lisp) use asymmetric dispatch, which avoids ambiguities by preferring certain dispatch arguments when searching for a method implementation.

Half & Half [Baumgartner et al. 2002] is a Java extension supporting asymmetric multiple dispatch but no external methods. Instead, it offers the ability to add new superinterfaces to existing classes, which requires structural conformance of the existing class with the new superinterface. JavaGI's retroactive interface implementations are more powerful because they can compensate for structural non-conformance by providing missing methods externally.

Nice [Bonniot et al. 2003] is a Java-like language supporting external methods and symmetric multiple dispatch. It has its roots in $\mathrm{ML}_{\leq}$ [Bourdoncle and Merz 1997], an explicitly typed extension of ML [Milner et al. 1997] with subtyping and higher-order multimethods. Nice also provides some form of retroactive interface implementation. Different from JavaGI, these retroactive implementations are not available for ordinary interfaces but only for so-called abstract interfaces. Unlike ordinary interfaces, abstract interfaces are not types but can be used to constrain type parameters [Bonniot 2003], in quite similar ways as JavaGI's implementation constraints.

Pirkelbauer et al. [2007] studied external methods and multiple dispatch in the context of C++. Their extension deals with the additional ambiguities arising through multiple inheritance by employing link-time checks. Allen et al. [2007] considered a formalization of external methods and multiple dispatch in the context of Fortress [Sun Microsystems 2008]. Their formalization includes multiple inheritance and defines modular restrictions that rule out ambiguous or undefined method calls.

An empirical study [Muschevici et al. 2008] analyzed the use of multiple dispatch in practice and suggested that "Java programs would have scope to use more multiple dispatch were it supported in the language." Along with other benefits, JavaGI offers an unobtrusive and compatible way of doing so without undue overhead.

7.6 Binary Methods

A binary method [Bruce et al. 1995] is a method that requires the receiver type and some of the argument types to coincide. Static typechecking of binary methods is challenging because subtyping treats method arguments contravariantly, whereas binary methods require arguments to vary covariantly.

PolyTOIL [Bruce et al. 2003] is a statically typed object-oriented languages supporting a keyword `MyType`, which represents the type of `this`. Using `MyType` as the type of certain method arguments provides faithful signatures for binary methods. To avoid the aforementioned tension between contra- and covariance, PolyTOIL separates subtyping from subclassing. Instead of subtyping, subclassing only induces a matching relation between types. Matching, written `<#`, is weaker than subtyping (i.e., relates more types) and can be used to constrain type parameters of classes and methods, leading to the notion of match-bounded polymorphism.

Although matching and subtyping are different relations, they are still quite similar. To avoid confusion between them, the successor $\mathcal{LOOM}$ [Bruce et al. 1997] of PolyTOIL completely eliminates subtyping in favor of matching. To address some loss of expressiveness, $\mathcal{LOOM}$ introduces hash types. A hash type `#T` denotes the set of all types matching `T`; that is, `#T` can be seen as an abbreviation for the match-bounded existential type $\exists$`X<#T.X.`

The language LOOJ [Bruce and Foster 2004] integrates the ideas of `MyType` into Java. It introduces `ThisClass` to capture the class type of `this` and `ThisType` to denote the public interface type of the definition where `ThisType` occurs. LOOJ ensures type safety in the presence of `ThisClass` and `ThisType` through exact types that essentially prohibit subtype polymorphism. Self-type constructors [Saito and Igarashi 2009] integrate the `ThisClass` construct of LOOJ with generics, so that `ThisClass` inside a generic class no longer denotes a specific instantiation of the class but takes type parameters on its own.

JavaGI provides explicit implementing types to express the signatures of binary methods in interfaces. To regain type soundness, JavaGI prevents invocations of binary methods on receivers whose static types are interface types and uses multiple dispatch to select the most specific implementation of a binary method dynamically. JavaGI also supports retroactive and constrained interface implementations, as well as static interface methods; these features have no correspondence in PolyTOIL, $\mathcal{LOOM}$, or LOOJ.

Eiffel's `like Current` construct [Meyer 1992] also allows one to express signatures of binary methods. Unfortunately, the construct renders the type system unsound [Cook 1989]. Attempts to recover type soundness include a global system validity check [Meyer 1992] and a complex condition preventing "polymorphic catcalls" [Meyer 1995].

Scala [Odersky 2009] supports singleton types of the form `this.type`, which are similar to (covariant uses of) `MyType` [Odersky and Zenger 2005b]. Moreover, Scala's self-type annotations allow programmers to state the type of `this` explicitly.

7.7 Type Conditionals

JavaGI's facility to provide methods and retroactive implementations of interfaces depending on the validity of type conditions is related to cJ [Huang et al. 2007],

a Java extension that provides type-conditional declarations of fields, methods, superclasses, and superinterfaces. JavaGI does not support type-conditional fields and superclasses. A type condition in cJ is any subtype constraint on generic parameters, whereas JavaGI also permits implementation constraints. The language cJ concentrates on type conditionals: it does not support JavaGI's retroactive implementations, multiple dispatch, explicit implementing types, multi-headed interfaces, and static interface methods.

Constraint-based polymorphism [Litvinov 1998; 2003] for Cecil [Chambers and the Cecil Group 2004] offers the possibility to define classes, subtype relationships, methods, and fields depending on certain type constraints. These constraints, expressed in `where`-clauses as in JavaGI, come in two forms: `isa`-constraints specify nominal subtype conditions, whereas `method`-constraints express structural subtype conditions. The system also supports external methods and multiple dispatch but does not provide an interface concept in the sense of JavaGI. The type system for constraint-based polymorphism is sound and there exists a sound and terminating but incomplete typechecking algorithm. In contrast, JavaGI's typechecking algorithm is sound, terminating, and complete, albeit for a weaker constraint system. Further, JavaGI is a conservative extension of the class-based language Java, whereas Cecil is an object-based language.

An extension of C# with type-equality constraints enables cast-free programs for object-oriented encodings of generalized algebraic datatypes [Kennedy and Russo 2005]. Further, it admits the specification of generic methods that only apply to certain instantiations of the enclosing class. While JavaGI does not support type equalities in their general form, it is nevertheless possible to encode several of the examples written with type equalities (e.g., the "typed expressions in typed environments" and the "list flatten" examples [Kennedy and Russo 2005]) using JavaGI's type conditionals. A generalization of type-equality constraints to arbitrary subtype constraints also includes variance for generic types [Emir et al. 2006].

As discussed in Section 7.4.7, Scala's views [Odersky 2009] can emulate some functionality of retroactive interface implementations. This functionality includes type-conditional interface implementations because views may place type conditions on their arguments.

Constrained types [Nystrom et al. 2008] in X10 [Saraswat 2009] are a form of dependent types [Pierce 2005, Chapter 2] that can enforce conditions on the immutable state of a class. This sort of condition is quite different from JavaGI's type conditions, which express subtype and implementation constraints on type variables.

The idea of separate `where`-clauses to specify type conditionals goes back to the programming language CLU [Liskov et al. 1981; Liskov et al. 1977]. CLU and its successor Theta [Liskov et al. 1995; Day et al. 1995] support structural constraints on type parameters, even if the parameters are defined in an enclosing scope.

7.8 Traits

Originally, a trait is a stateless collection of methods implementing a particular concern, but separate from a class [Ducasse et al. 2006; Ducasse 2009]. Traits can be composed in various ways and have to be included in a class to attach their methods to objects of that class. Recent work also addresses stateful traits

[Bergel et al. 2008] and integrates traits into statically typed languages [Smith and Drossopoulou 2005; Liquori and Spiwack 2008; Bono et al. 2008]. The main difference between traits and generalized interfaces in JavaGI is the intention behind these two concepts: traits are meant as units of reuse whereas interfaces describe signatures of objects.

Scala [Odersky 2009] combines these two intentions. Like interfaces in Java, traits specify signatures of objects but they may also contain fields and default implementations of certain methods. JavaGI simulates such default implementations with implementation inheritance and abstract implementation definitions. Modular mixin composition [Odersky and Zenger 2005b] integrates traits into classes. Unlike JavaGI, however, Scala does not support retroactive implementations of traits. Traits in Fortress [Sun Microsystems 2008] are like Java interfaces but they may contain code, properties, and include parameterization over values.

Mohnen [2002] suggested interfaces with default implementations for Java. JavaGI can encode such default implementations with abstract implementations and implementation inheritance (see Section 2.5).

7.9 Advanced Subtyping Mechanisms

This section discusses some advanced subtyping mechanisms related to JavaGI.

Most object-oriented languages (e.g., Java, C#, Scala, and also JavaGI) rely on nominal subtyping, where explicit declarations establish the subtyping relation. In contrast, structural subtyping considers type T a subtype of another type U if T matches U structurally; that is, T supports at least the features provided by U. Structural subtyping enables retroactive interface implementation if the names and signatures of the methods of a class happen to match the requirements of an interface. In practice, however, situations where a class does not implement an interface nominally but nevertheless provides all the interface's methods with exactly matching signatures seem to be quite rare. It appears to be more common that a class provides the methods of an interface with slight mismatches with respect to method names or argument ordering [Hölzle 1993]. Structural subtyping alone does not help in such situations but JavaGI's retroactive interface implementation does. Nevertheless, structural subtyping provides benefits in other problem areas [Malayeri and Aldrich 2009]. Ostermann [Ostermann 2008] provides a detailed comparison between nominal and structural subtyping.

Compound types [Büchi and Weck 1998] integrate a form of intersection types [Pierce 2002, Section 15.7] into Java. They are subject to structural subtyping, but other constructs of the language still rely on nominal subtyping. Läufer et al. [2000] consider structural conformance to interface types in the context of Java. In their work, a type is a subtype of some interface if it matches the interface structurally. The authors also discuss a renaming mechanism for methods to make structural conformance more widely applicable. Whiteoak [Gil and Maman 2008] is an extension of Java that introduces designated `struct` types. These types are subject to structural subtyping and support flexible composition operations. Unity [Malayeri and Aldrich 2008] is a language design that integrates nominal and structural subtyping, and also provides external methods.

The programming language Sather [Szyperski et al. 1994; Stoutamire and Omohundro 1996] is based on nominal subtyping but recovers some of the flexibility of

structural subtyping by supporting not only declarations of subtype but also of supertypes. Further, Sather decouples inheritance from subtyping [Cook et al. 1990]. The calculus $FJ_{<:}$ [Ostermann 2008] combines Sather's subtyping mechanism with compound types [Büchi and Weck 1998] to arrive at a non-transitive subtyping relation. Pedersen [1989] proposed specialization (viz., the possibility to introduce new superclasses) as a technique to improve reusability of classes.

8. CONCLUSION AND FUTURE WORK

JavaGI is a conservative extension of Java based on the notion of generalized interfaces. It offers a flexible approach to adapting, extending, and integrating existing software components, even in binary form. Further, JavaGI supersedes tedious applications of design patterns and offers save and convenient alternatives to unsafe cast operations, run-time exceptions, and code duplication. The generalization of interfaces serves as the unifying notion that leads to a coherent and elegant language design. Thus, JavaGI smoothly integrates features only loosely connected in other language proposals.

Future work addresses support for associated types and proper reflection facilities. Furthermore, the following directions may be worthwhile to pursue.

Implementation Families. Currently, all retroactive implementation definitions share a global scope. This approach may lead to problems composing separately developed parts of an application because it impedes independent extensibility [Szyperski 1996]. For example, different parts of an application may need to provide different implementations of the same interface with identical implementing types. Unfortunately, JavaGI prevents such overlapping implementations to rule out ambiguities in dynamic method lookup. *Implementation families* are a possible solution to this problem. The idea is to partition the set of implementation definitions into disjoint families such that JavaGI's global well-formedness criteria must hold only within each family and not for all implementation definitions. To avoid run-time ambiguities, an invocation of a retroactively implemented method must specify, either explicitly or implicitly, the family from which to resolve the implementation.

Better Support for Interfaces as Implementing Types. Currently, JavaGI prevents retroactive implementations of interfaces that are used as implementing types in other implementations (Section 3.4.5). Again, this restriction endangers independent extensibility. It is an open question how to lift the restriction in a satisfactory manner. On the theoretical side, the restriction is important to ensure decidability of constraint entailment and subtyping [Wehr and Thiemann 2009b]. On the practical side, the restriction enables efficient run-time lookup of retroactive implementations.

Retroactive Interface Implementations for the Java Virtual Machine. Currently, the JavaGI compiler generates code that is executable on an unmodified Java Virtual Machine (JVM [Lindholm and Yellin 1999]). It would be worthwhile to explore what modifications to the JVM are necessary to support retroactive interface implementations directly. Possible benefits of such an extension include better performance and improved compatibility with libraries compiled by an ordinary Java compiler. (Libraries compiled by an ordinary Java compiler are not aware of wrappers and

thus may exhibit unexpected behavior under the current compilation approach.)

Generalized Interfaces for C#. Currently, generalized interfaces are only available as an extension of the language Java. What about generalized interfaces for other object-oriented languages such as C#? Although Java and C# are quite similar, there are still sufficiently many differences that would make such an undertaking interesting. For example, Java has a type-erasure semantics; that is, type arguments of generic classes are not available at run time. In contrast, C# provides run-time types. As discussed in Section 5.1.3, Java's type-erasure semantics influenced the design of JavaGI at several places, so the availability of run-time types may require rethinking some of these design decisions.

REFERENCES

Allen, E., Hallett, J. J., Luchangco, V., Ryu, S., and Steele Jr., G. L. 2007. Modular multiple dispatch with multiple inheritance. In *ACM Symposium on Applied Computing (SAC)*, pp. 1117–1121. ACM Press.

Ancona, D. and Zucca, E. 2001. True modules for Java-like languages. See ECOOP [2001], pp. 354–380.

Apache Software Foundation 2009. Apache Tomcat. `http://tomcat.apache.org/`.

Apple Inc 2009. The Objective-C programming language. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

AspectJ Team 2009a. The AspectJ development environment guide. `http://www.eclipse.org/aspectj/doc/released/devguide/index.html`.

AspectJ Team 2009b. The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press.

Baumgartner, G., Jansche, M., and Läufer, K. 2002. Half & Half: Multiple dispatch and retroactive abstraction for Java. Tech. Rep. OSU-CISRC-5/01-TR08, Revised 3/02, Ohio State University. `http://www.csc.lsu.edu/~gb/Brew/Publications/HalfNHalf.pdf`.

Bergel, A., Ducasse, S., and Nierstrasz, O. 2005. Classbox/J: Controlling the scope of change in Java. See OOPSLA [2005], pp. 177–189.

Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R. 2005. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems & Structures 31,* 3–4, 107–126.

Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R. 2008. Stateful traits and their formalization. *Computer Languages, Systems & Structures 34,* 2–3, 83–108.

Bergel, A., Ducasse, S., and Wuyts, R. 2003. Classboxes: A minimal module model supporting local rebinding. In *Joint Modular Languages Conference (JMLC)*, Volume 2789 of *Lecture Notes in Computer Science*, pp. 122–131. Springer-Verlag.

Bernardy, J.-P., Jansson, P., Zalewski, M., Schupp, S., and Priesnitz, A. 2008. A comparison of C++ concepts and Haskell type classes. In *ACM SIGPLAN Workshop on Generic Programming*, pp. 37–48. ACM Press.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. See OOPSLA [2006], pp. 169–190.

Bonniot, D. 2003. Using kinds to type partially-polymorphic methods. *Electronic Notes in Theoretical Computer Science 75*, 21–40.

Bonniot, D., Keller, B., and Barber, F. 2003. The Nice user's manual. `http://nice.sourceforge.net/manual.html`.

Bono, V., Damiani, F., and Giachino, E. 2008. On traits and types in a Java-like setting. In *IFIP International Conference On Theoretical Computer Science (TCS)*, pp. 367–382. Springer-Verlag.

Bourdoncle, F. and Merz, S. 1997. Type checking higher-order polymorphic multi-methods. See POPL [1997], pp. 302–315.

Bracha, G. 2004. Generics in the Java programming language. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. See OOPSLA [1990], pp. 303–311.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. See OOPSLA [1998], pp. 183–200.

Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T., and Pierce, B. C. 1995. On binary methods. *Theory and Practice of Object Systems 1,* 3, 221–242.

Bruce, K. B. and Foster, J. N. 2004. LOOJ: Weaving LOOM into Java. See ECOOP [2004], pp. 389–413.

Bruce, K. B., Odersky, M., and Wadler, P. 1998. A statically safe alternative to virtual types. See ECOOP [1998], pp. 523–549.

Bruce, K. B., Petersen, L., and Fiech, A. 1997. Subtyping is not a good "match" for object-oriented languages. See ECOOP [1997], pp. 104–127.

Bruce, K. B., Schuett, A., and van Gent, R. 1995. PolyTOIL: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 952 of *Lecture Notes in Computer Science*, pp. 27–51. Springer-Verlag.

Bruce, K. B., Schuett, A., van Gent, R., and Fiech, A. 2003. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems 25,* 2, 225–290.

Büchi, M. and Weck, W. 1998. Compound types for Java. See OOPSLA [1998], pp. 362–373.

Cameron, N., Drossopoulou, S., and Ernst, E. 2008. A model for Java with wildcards. See ECOOP [2008], pp. 2–26.

Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J. C. 1989. F-bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 273–280. ACM Press.

Chakravarty, M. M. T., Keller, G., and Peyton Jones, S. 2005. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 241–253. ACM Press.

Chakravarty, M. M. T., Keller, G., Peyton Jones, S., and Marlow, S. 2005. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 1–13. ACM Press.

Chambers, C. 1992. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 615 of *Lecture Notes in Computer Science*, pp. 33–56. Springer-Verlag.

Chambers, C. and Leavens, G. T. 1996. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. Technical Report TR-96-12-02, University of Washington, Department of Computer Science and Engineering.

Chambers, C. and the Cecil Group 2004. The Cecil language: Specification and rationale, version 3.2. `http://www.cs.washington.edu/research/projects/cecil/pubs/cecil-spec.html`.

Clark, J. and DeRose, S. 1999. XML path language (XPath), version 1.0. `http://www.w3.org/TR/xpath`.

Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. See OOPSLA [2000], pp. 130–145.

Clifton, C., Millstein, T., Leavens, G. T., and Chambers, C. 2006. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems 28,* 3, 517–575.

Cook, W. R. 1989. A proposal for making Eiffel type-safe. See ECOOP [1989], pp. 57–70.

Cook, W. R. 1991. Object-oriented programming versus abstract data types. In *REX School/-Workshop on Foundations of Object-Oriented Languages*, Volume 489 of *Lecture Notes in Computer Science*, pp. 151–178. Springer-Verlag.

Cook, W. R., Hill, W., and Canning, P. S. 1990. Inheritance is not subtyping. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 125–135. ACM Press.

Day, M., Gruber, R., Liskov, B., and Myers, A. C. 1995. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 156–168. ACM Press.

dom4j 2008. Dom4j — An open source XML framework for Java. `http://www.dom4j.org/`.

Ducasse, S. 2009. Putting traits in perspective. In *International Conference on Software Engineering (ICSE)*, Volume 5634 of *Lecture Notes in Computer Science*, pp. 5–8. Springer-Verlag.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. 2006. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems 28,* 2, 331–388.

Eclipse 2009. Eclipse — An open development platform. `http://www.eclipse.org/`.

Eclipse Foundation 2004. Eclipse public license. `http://www.eclipse.org/legal/epl-v10.html`.

Eclipse Foundation 2008. Eclipse compiler for Java. `http://download.eclipse.org/eclipse/downloads/drops/R-3.4.1-200809111700/index.php`.

ECMA International 2002. Standard 334: C# language specification, 2nd edition. `http://www.ecma-international.org/publications/standards/Ecma-334-arch.htm`.

ECMA International 2005. Standard 334: C# language specification, 3rd edition. `http://www.ecma-international.org/publications/standards/Ecma-334-arch.htm`.

ECMA International 2006. Standard 335: Common language infrastructure, 4th edition. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

ECOOP 1989. *European Conference on Object-Oriented Programming (ECOOP)*. Cambridge University Press.

ECOOP 1997. *European Conference on Object-Oriented Programming (ECOOP)*, Volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag.

ECOOP 1998. *European Conference on Object-Oriented Programming (ECOOP)*, Volume 1445 of *Lecture Notes in Computer Science*. Springer-Verlag.

ECOOP 2001. *European Conference on Object-Oriented Programming (ECOOP)*, Volume 2072 of *Lecture Notes in Computer Science*. Springer-Verlag.

ECOOP 2004. *European Conference on Object-Oriented Programming (ECOOP)*, Volume 3086 of *Lecture Notes in Computer Science*. Springer-Verlag.

ECOOP 2008. *European Conference on Object-Oriented Programming (ECOOP)*, Volume 5142 of *Lecture Notes in Computer Science*. Springer-Verlag.

Emir, B., Kennedy, A., Russo, C. V., and Yu, D. 2006. Variance and generalized constraints for C# generics. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 4067 of *Lecture Notes in Computer Science*, pp. 279–303. Springer-Verlag.

Ernst, E. 1999. gbeta – a language with virtual attributes, block structure, and propagating, dynamic inheritance. Ph.D. thesis, Department of Computer Science, University of Åarhus, Denmark.

Ernst, E. 2001. Family polymorphism. See ECOOP [2001], pp. 303–326.

Ernst, E. 2003. Higher-order hierarchies. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 2743 of *Lecture Notes in Computer Science*, pp. 303–329. Springer-Verlag.

Flatt, M. and Felleisen, M. 1998. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 236–248. ACM Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., and Willcock, J. 2007. An extended comparative study of language support for generic programming. *Journal of Functional Programming 17,* 02, 145–205.

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J. G., and Willcock, J. 2003. A comparative study of language support for generic programming. See OOPSLA [2003], pp. 115–134.

Gasiunas, V., Mezini, M., and Ostermann, K. 2007. Dependent classes. See OOPSLA [2007], pp. 133–152.

Gil, J. and Maman, I. 2008. Whiteoak: Introducing structural typing into Java. See OOPSLA [2008], pp. 73–90.

Girard, J.-Y. 1972. Interpretation fonctionnelle et elimination des coupures dans l'arithmetique d'ordre superieur. Ph.D. thesis, University of Paris VII.

Goldberg, A. and Robson, D. 1989. *Smalltalk 80: The Language*. Addison-Wesley.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java Language Specification* (3rd ed.). Addison-Wesley.

GPCE 2007. *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM Press.

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A. 2006. Concepts: Linguistic support for generic programming in C++. See OOPSLA [2006], pp. 291–310.

Hall, C. V., Hammond, K., Peyton Jones, S. L., and Wadler, P. L. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems 18,* 2, 109–138.

Harrison, W. and Ossher, H. 1993. Subject-oriented programming: A critique of pure objects. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 411–428. ACM Press.

Haskell Workshop 2004. *ACM SIGPLAN Haskell Workshop*.

Helm, R., Holland, I. M., and Gangopadhyay, D. 1990. Contracts: Specifying behavioral compositions in object-oriented systems. See OOPSLA [1990], pp. 169–180.

Hölzle, U. 1993. Integrating independently-developed components in object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 707 of *Lecture Notes in Computer Science*, pp. 36–56. Springer-Verlag.

Huang, S. S., Zook, D., and Smaragdakis, Y. 2007. cJ: Enhancing Java with safe type conditions. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 185–198. ACM Press.

Hummel, O. and Atkinson, C. 2009. The managed adapter pattern: Facilitating glue code generation for component reuse. In *International Conference on Software Reuse (ICSR)*, pp. 211–224. Springer-Verlag.

Hunter, J. and McLaughlin, B. 2007. JDOM. `http://www.jdom.org/`.

ICSE 1999. *International Conference on Software Engineering (ICSE)*. ACM Press.

Igarashi, A., Pierce, B. C., and Wadler, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23,* 3, 396–450.

Igarashi, A. and Viroli, M. 2007. Variant path types for scalable extensibility. See OOPSLA [2007], pp. 113–132.

Ingalls, D. H. H. 1986. A simple technique for handling multiple polymorphism. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

Järvi, J., Willcock, J., and Lumsdaine, A. 2003. Concept-controlled polymorphism. In *International Conference on Generative Programming and Component Engineering (GPCE)*, Volume 2830 of *Lecture Notes in Computer Science*, pp. 228–244. Springer-Verlag.

Järvi, J., Willcock, J., and Lumsdaine, A. 2005. Associated types and constraint propagation for mainstream object-oriented generics. See OOPSLA [2005], pp. 1–19.

Jaxen 2008. Jaxen — A universal Java XPath engine. `http://jaxen.codehaus.org/`.

Jones, M. P. 1993. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 52–61. ACM Press.

Jones, M. P. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press.

Jones, M. P. 2000. Type classes with functional dependencies. In *European Symposium on Programming (ESOP)*, Volume 1782 of *Lecture Notes in Computer Science*, pp. 230–244. Springer-Verlag.

Kaes, S. 1988. Parametric overloading in polymorphic programming languages. In *European Symposium on Programming (ESOP)*, Volume 300 of *Lecture Notes in Computer Science*, pp. 131–144. Springer-Verlag.

Kahl, W. and Scheffczyk, J. 2001. Named instances for Haskell type classes. In *Haskell Workshop*.

Kamina, T. and Tamai, T. 2007. Lightweight scalable components. See GPCE [2007], pp. 145–154.

Kamina, T. and Tamai, T. 2008. Lightweight dependent classes. In *ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, pp. 113–124. ACM Press.

Keller, R. and Hölzle, U. 1998. Binary component adaptation. See ECOOP [1998], pp. 307–329.

Kennedy, A. and Russo, C. 2005. Generalized algebraic data types and object-oriented programming. See OOPSLA [2005], pp. 21–40.

Kennedy, A. and Syme, D. 2001. Design and implementation of generics for the .NET common language runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–12. ACM Press.

Kennedy, A. J. and Pierce, B. C. 2007. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD),* informal proceedings. `http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html`.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An overview of AspectJ. See ECOOP [2001], pp. 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. See ECOOP [1997], pp. 220–242.

Kiselyov, O. and Lämmel, R. 2005. Haskell's overlooked object system. `http://homepages.cwi.nl/~ralf/OOHaskell/`.

Kiselyov, O., Lämmel, R., and Schupke, K. 2004. Strongly typed heterogeneous collections. See Haskell Workshop [2004], pp. 96–107.

Kiselyov, O. and Shan, C.-c. 2004. Functional pearl: Implicit configurations—or, type classes reflect the values of types. See Haskell Workshop [2004], pp. 33–44.

Lämmel, R. and Ostermann, K. 2006. Software extension and integration with type classes. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pp. 161–170. ACM Press.

Läufer, K., Baumgartner, G., and Russo, V. F. 2000. Safe structural conformance for Java. *The Computer Journal 43,* 6, 469–481.

Leavens, G. T. and Millstein, T. D. 1998. Multiple dispatch as dispatch on tuples. See OOPSLA [1998], pp. 374–387.

Leroy, X. 2008. The Objective Caml system release 3.11. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

Lindholm, T. and Yellin, F. 1999. *The Java Virtual Machine Specification* (2nd ed.). Addison-Wesley.

Liquori, L. and Spiwack, A. 2008. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems 30,* 2, 1–32.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J. C., Scheifler, R., and Snyder, A. 1981. *CLU reference manual*, Volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag.

Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P., and Myers, A. C. 1995. Theta reference manual, preliminary version. `http://www.pmg.csail.mit.edu/papers/thetaref.ps.gz`.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction mechanisms in CLU. *Communications of the ACM 20,* 8, 564–576.

Litvinov, V. 1998. Contraint-based polymorphism in Cecil: Towards a practical and static type system. See OOPSLA [1998], pp. 388–411.

Litvinov, V. 2003. Constraint-bounded polymorphism: An expressive and practical type system for object-oriented languages. Ph.D. thesis, University of Washington.

Madsen, O. and Møller-Pedersen, B. 1989. Virtual classes: A powerful mechanism in object-oriented programming. See OOPSLA [1989], pp. 397–406.

Malayeri, D. and Aldrich, J. 2008. Integrating nominal and structural subtyping. See ECOOP [2008], pp. 260–284.

Malayeri, D. and Aldrich, J. 2009. Is structural subtyping useful? An empirical study. In *European Symposium on Programming (ESOP)*, Volume 5502 of *Lecture Notes in Computer Science*, pp. 95–111. Springer-Verlag.

Mattsson, M., Bosch, J., and Fayad, M. E. 1999. Framework integration problems, causes, solutions. *Communications of the ACM 42,* 10, 80–87.

Mazurak, K. and Zdancewic, S. 2006. Type inference for Java 5: Wildcards, F-bounds, and undecidability. `http://www.cis.upenn.edu/~stevez/note.html`.

McDirmid, S., Flatt, M., and Hsieh, W. C. 2001. Jiazzi: New-age components for old-fasioned Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 211–222. ACM Press.

McNamara, B. and Smaragdakis, Y. 2000. Static interfaces in C++. In *Workshop on C++ Template Programming,* informal proceedings. `http://www.oonumerics.org/tmpw00/mcnamara.pdf`.

Meyer, B. 1992. *Eiffel: The Language*. Prentice-Hall.

Meyer, B. 1995. Static typing. *ACM SIGPLAN OOPS Messenger 6,* 4, 20–29.

Mezini, M. and Ostermann, K. 2002. Integrating independent components with on-demand remodularization. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 52–67. ACM Press.

Mezini, M., Seiter, L., and Lieberherr, K. 2000. Component integration with pluggable composite adapters. In M. Aksit (Ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers.

Millstein, T., Reay, M., and Chambers, C. 2003. Relaxed MultiJava: Balancing extensibility and modular typechecking. See OOPSLA [2003], pp. 224–240.

Millstein, T. D. and Chambers, C. 1999. Modular statically typed multimethods. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 1628 of *Lecture Notes in Computer Science*, pp. 279–303. Springer-Verlag.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Mohnen, M. 2002. Interfaces with default implementations in Java. In *Conference on the Principles and Practice of Programming in Java (PPPJ)*, pp. 35–40. ACM Press.

Moors, A., Piessens, F., and Odersky, M. 2008. Generics of a higher kind. See OOPSLA [2008], pp. 423–438.

Muschevici, R., Potanin, A., Tempero, E., and Noble, J. 2008. Multiple dispatch in practice. See OOPSLA [2008], pp. 563–582.

Myers, A. C., Bank, J. A., and Liskov, B. 1997. Parameterized types for java. See POPL [1997], pp. 132–145.

Myers, N. 1996. A new and useful template technique: "traits". In S. B. Lippman (Ed.), *C++ gems*, pp. 451–457. SIGS Publications, Inc.

MzScheme 2009. MzScheme — Core virtual machine for PLT Scheme. `http://www.plt-scheme.org/software/mzscheme/`.

Nystrom, N., Chong, S., and Myers, A. C. 2004. Scalable extensibility via nested inheritance. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 99–115. ACM Press.

Nystrom, N., Qi, X., and Myers, A. C. 2006. J&: Nested intersection for scalable software composition. See OOPSLA [2006], pp. 21–36.

Nystrom, N., Saraswat, V., Palsberg, J., and Grothoff, C. 2008. Constrained types for object-oriented languages. See OOPSLA [2008], pp. 457–474.

Odersky, M. 2009. The Scala language specification, version 2.7. Draft, `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

Odersky, M. and Zenger, M. 2005a. Independently extensible solutions to the expression problem. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, informal proceedings. `http://homepages.inf.ed.ac.uk/wadler/fool/program/10.html`.

Odersky, M. and Zenger, M. 2005b. Scalable component abstractions. See OOPSLA [2005], pp. 41–58.

OOPSLA 1989. *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 1990. *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*. ACM Press.

OOPSLA 1998. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 2000. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 2003. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 2005. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 2006. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 2007. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

OOPSLA 2008. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press.

Ossher, H. and Tarr, P. 1999. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. See ICSE [1999], pp. 687–688.

Ossher, H. and Tarr, P. 2000. Hyper/J: Multi-dimensional separation of concerns for Java. In *International Conference on Software Engineering (ICSE)*, pp. 734–737. ACM Press.

Ostermann, K. 2008. Nominal and structural subtyping in component-based programming. *Journal of Object Technology 7,* 1, 121–145. `http://www.jot.fm/issues/issue_2008_01/article4/`.

Pedersen, C. H. 1989. Extending ordinary inheritance schemes to include generalization. See OOPSLA [1989], pp. 407–417.

Peyton Jones, S. (Ed.) 2003. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press.

Peyton Jones, S., Jones, M., and Meijer, E. 1997. Type classes: An exploration of the design space. In *Haskell Workshop*.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

Pierce, B. C. (Ed.) 2005. *Advanced Topics in Types and Programming Languages*. MIT Press.

Pirkelbauer, P., Solodkyy, Y., and Stroustrup, B. 2007. Open multi-methods for C++. See GPCE [2007], pp. 123–134.

Plotkin, G. 1981. A structural approach to operational semantics. Technical Report DAIMI FN-19, Åarhus University, Denmark.

POPL 1997. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press.

Qi, X. and Myers, A. C. 2009. Sharing classes between families. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 281–292. ACM Press.

Reis, G. D. and Stroustrup, B. 2006. Specifying C++ concepts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 295–308. ACM Press.

Rémy, D. and Vouillon, J. 1998a. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems 4,* 1, 27–50.

Rémy, D. and Vouillon, J. 1998b. On the (un)reality of virtual types. `http://gallium.inria.fr/~remy/work/virtual/virtual.ps.gz`.

Reynolds, J. C. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, Volume 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag.

Reynolds, J. C. 1975. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schumann (Ed.), *New Directions in Algorithmic Languages*. INRIA. Reprinted in [Reynolds 1994].

Reynolds, J. C. 1994. User-defined types and procedural data structures as complementary approaches to data abstraction. In C. A. Gunter and J. C. Mitchell (Eds.), *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pp. 13–23. MIT Press. Originally published in [Reynolds 1975].

Rémy, D. and Vouillon, J. 1997. Objective ML: A simple object-oriented extension of ML. See POPL [1997], pp. 40–53.

Saito, C. and Igarashi, A. 2009. Self type constructors. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 263–282. ACM Press.

Saito, C., Igarashi, A., and Viroli, M. 2008. Lightweight family polymorphism. *Journal of Functional Programming 18,* 3, 285–331.

Sakkinen, M. 1989. Disciplined inheritance. See ECOOP [1989], pp. 39–56.

Saraswat, V. 2009. Report on the programming language X10, version 2.0. `http://dist.codehaus.org/x10/documentation/languagespec/x10-200.pdf`.

Sasitorn, J. and Cartwright, R. 2007. Component NextGen: A sound and expressive component framework for Java. See OOPSLA [2007], pp. 153–170.

Sekharaiah, K. C. and Ram, D. J. 2002. Object schizophrenia problem in object role system design. In *International Conference on Object-Oriented Information Systems (OOIS)*, pp. 494–506. Springer-Verlag.

Shalit, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Programming Language*. Addison-Wesley.

Siek, J. and Lumsdaine, A. 2005. Essential language support for generic programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 73–84. ACM Press.

Siek, J. G. 2005. A language for generic programming. Ph.D. thesis, Indiana University.

Siek, J. G., Lee, L.-Q., and Lumsdaine, A. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley.

Smith, C. and Drossopoulou, S. 2005. Chai: Typed traits in java. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 3586 of *Lecture Notes in Computer Science*, pp. 543–576. Springer-Verlag.

Smith, D. and Cartwright, R. 2008. Java type inference is broken: Can we fix it? See OOPSLA [2008], pp. 505–524.

Steele, G. 1990. *Common LISP: The Language* (2nd ed.). Digital Press.

Stepanov, A. and Lee, M. 1995. The standard template library. Tech. rep., WG21/N0482, ISO Programming Language C++ Project.

Stoutamire, D. and Omohundro, S. 1996. The Sather 1.1 specification. Tech. Rep. TR-96-012, International Computer Science Institute.

Strniša, R., Sewell, P., and Parkinson, M. 2007. The Java module system: Core design and semantic definition. See OOPSLA [2007], pp. 499–514.

Sulzmann, M. 2006. Extracting programs from type class proofs. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, pp. 97–108. ACM Press.

Sulzmann, M., Duck, G. J., Peyton Jones, S., and Stuckey, P. J. 2007. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming 17,* 1, 83–129.

Sun Microsystems 2004a. The collections framework. `http://java.sun.com/j2se/1.5.0/docs/guide/collections/`.

Sun Microsystems 2004b. Java 2 platform standard edition 5.0 API specification. `http://java.sun.com/j2se/1.5.0/docs/api/index.html`.

Sun Microsystems 2006. JSR 277: Java module system. `http://jcp.org/en/jsr/detail?id=277`.

Sun Microsystems 2007. Java servlet specification, version 2.5. `http://java.sun.com/products/servlet/`.

Sun Microsystems 2008. Project Fortress website. `http://projectfortress.sun.com/`.

Sun Microsystems 2009. Java platform standard edition. `http://java.sun.com/javase/`.

Szyperski, C. 1996. Independently extensible systems — Software engineering potential and challenges. In *Australasian Computer Science Conference (ACSC)*.

Szyperski, C., Omohundro, S., and Murer, S. 1994. Engineering a programming language: The type and class system of Sather. In *International Conference on Programming Languages and Systems Architecture*, Volume 782 of *Lecture Notes in Computer Science*, pp. 208–227. Springer-Verlag.

Tarr, P., Ossher, H., Harrison, W., and Jr., S. M. S. 1999. *N* degrees of separation: Multi-dimensional separation of concerns. See ICSE [1999], pp. 107–119.

Thiemann, P. 2005. An embedded domain-specific language for type-safe server-side Web-scripting. *ACM Transactions on Internet Technology 5,* 1, 1–46.

Thiemann, P. and Wehr, S. 2008. Interface types for Haskell. In *Asian Symposium on Programming Languages and Systems (APLAS)*, Volume 5356 of *Lecture Notes in Computer Science*, pp. 256–272. Springer-Verlag.

Torgersen, M. 2004. The expression problem revisited — Four new solutions using generics. See ECOOP [2004], pp. 123–143.

Torgersen, M., Ernst, E., Hansen, C. P., von der Ahé, P., Bracha, G., and Gafter, N. 2004. Adding wildcards to the Java programming language. *Journal of Object Technology 3,* 11, 97–116. `http://www.jot.fm/issues/issue_2004_12/article5/`.

Trifonov, V. and Smith, S. 1996. Subtyping constrained types. In *International Symposium on Static Analysis (SAS)*, Volume 1145 of *Lecture Notes in Computer Science*, pp. 349–365. Springer-Verlag.

Viroli, M. 2000. On the recursive generation of parametric types. Tech. Rep. DEIS-LIA-00-002, Università di Bologna.

Viroli, M. and Natali, A. 2000. Parametric polymorphism in Java: An approach to translation based on reflective features. See OOPSLA [2000], pp. 146–165.

Wadler, P. 1998. The expression problem. Post to the Java Genericity mailing list.

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad-hoc. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 60–76. ACM Press.

Warth, A., Stanojevic, M., and Millstein, T. 2006. Statically scoped object adaptation with expanders. See OOPSLA [2006], pp. 37–56.

Wehr, S. 2005. Problem with superclass entailment in "A Static Semantics for Haskell". Post to the Haskell mailinglist, `http://www.haskell.org//pipermail/haskell/2005-October/016695.html`.

Wehr, S. 2009. JavaGI homepage. `http://www.informatik.uni-freiburg.de/~wehr/javagi`.

Wehr, S. 2010. JavaGI: A language with generalized interfaces. Ph.D. thesis, Technische Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau, Germany. URL: `http://www.freidok.uni-freiburg.de/volltexte/7678/`, URN: `urn:nbn:de:bsz:25-opus-76785`.

Wehr, S., Lämmel, R., and Thiemann, P. 2007. JavaGI: Generalized interfaces for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 4609 of *Lecture Notes in Computer Science*, pp. 347–372. Springer-Verlag.

Wehr, S. and Thiemann, P. 2008. Subtyping existential types. In *Workshop on Formal Techniques for Java-like Programs (FTfJP),* informal proceedings, pp. 125–136. `http://www-sop.inria.fr/everest/events/FTfJP08/ftfjp08.pdf`.

Wehr, S. and Thiemann, P. 2009a. JavaGI in the battlefield: Practical experience with generalized interfaces. In *ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, pp. 65–74. ACM Press.

Wehr, S. and Thiemann, P. 2009b. On the decidability of subtyping with bounded existential types. In *Asian Symposium on Programming Languages and Systems (APLAS)*.

Wright, A. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.

Yu, D., Kennedy, A., and Syme, D. 2004. Formalization of generics for the .NET common language runtime. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 39–51. ACM Press.

Zenger, M. 2005. Keris: Evolving software with extensible modules. *Journal of Software Maintenance and Evolution: Research and Practice 17,* 5, 333–362.

A. SYNTAX OF JAVAGI

Figures 30 and 31 define the syntax of JavaGI, expressed as an extension to the syntax of Java as defined in the first 17 chapters of *The Java Language Specification* (JLS) [Gosling et al. 2005]. The syntax definition shows nonterminal symbols in *italic* font and terminal symbols in `fixed width` font. The subscript "*opt*" indicates an optional item. Alternative productions for the same nonterminal are separated by the symbol "|". A nonterminal already defined in the JLS carries a superscript annotation specifying the JLS section of its original definition. A JLS section annotation in parenthesis indicates that the syntax of JavaGI redefines the annotated nonterminal. An ellipsis "..." represents unmodified JLS productions. The figure highlights changes to other productions from the JLS.

There are three new keywords in JavaGI: `implementation`, `receiver`, and `where`. The nonterminal `as` in the production for *ImplName* in Figure 30 is not parsed as a keyword but as a regular identifier.

Implementations

TypeDeclaration(§ 7.6) : ... | *ImplDeclaration*

ImplDeclaration : *ImplModifier*$_{opt}$ `implementation` *TypeParameters*$^{\S 8.1.2}_{opt}$ *InterfaceType*§ 4.3 `[` *ClassOrInterfaceTypeList* `]` *ImplName*$_{opt}$ *ExtendsImpl*$_{opt}$ *ConstraintClause*$_{opt}$ `{` *ImplBodyDeclarations*$_{opt}$ `}`

ImplModifier : one of `final` `abstract`

ClassOrInterfaceTypeList : non-empty list of *ClassOrInterfaceType*§ 4.3 separated by `,`

ImplName : `as` *Identifier*§ 3.8

ExtendsImpl : `extends` *ImplReference*

ImplReference : *InterfaceType*§ 4.3 `[` *ClassOrInterfaceTypeList* `]` | *TypeName*§ 4.3

ConstraintClause : `where` *ConstraintList*

ConstraintList : non-empty list of *Constraint* separated by `,`

Constraint : *ReferenceType*§ 4.3 *TypeBound*(§ 4.4) | *ImplTypeList* `implements` *InterfaceType*§ 4.3

ImplTypeList : non-empty list of *ReferenceType*§ 4.3 separated by `*`

ImplBodyDeclarations : possibly empty list of *ImplBodyDeclaration*

ImplBodyDeclaration : *MethodDeclaration*§ 8.4 | *ReceiverImpl*

ReceiverImpl : `receiver` *ClassOrInterfaceType*§ 4.3 `{` *MethodDeclarations*$_{opt}$ `}`

MethodDeclarations : possibly empty list of *MethodDeclaration*§ 8.4

Interfaces

NormalInterfaceDeclaration(§ 9.1) : *InterfaceModifiers*$^{\S 9.1.1}_{opt}$ `interface` *Identifier*§ 3.8 *TypeParameters*$^{\S 8.1.2}_{opt}$ *ImplParameters*$_{opt}$ *ExtendsInterfaces*$^{\S 9.1.3}_{opt}$ *ConstraintClause*$_{opt}$ *InterfaceBody*§ 9.1.4

ImplParameters : `[` *IdentifierList* *ConstraintClause*$_{opt}$ `]`

IdentifierList : non-empty list of *Identifier*§ 3.8 separated by `,`

InterfaceMemberDeclaration(§ 9.1.4) : ... | *ReceiverDeclaration*

ReceiverDeclaration : `receiver` *Identifier*§ 3.8 `{` *AbstractMethodDeclarations*$^{\S 9.4}_{opt}$ `}`

Classes

NormalClassDeclaration(§ 8.1) : *ClassModifiers*$^{\S 8.1.1}_{opt}$ `class` *Identifier*§ 3.8 *TypeParameters*$^{\S 8.1.2}_{opt}$ *Super*$^{\S 8.1.4}_{opt}$ *Interfaces*$^{\S 8.1.5}_{opt}$ *ConstraintClause*$_{opt}$ *ClassBody*§ 8.1.6

Figure 30: Syntax of JavaGI (1/2).

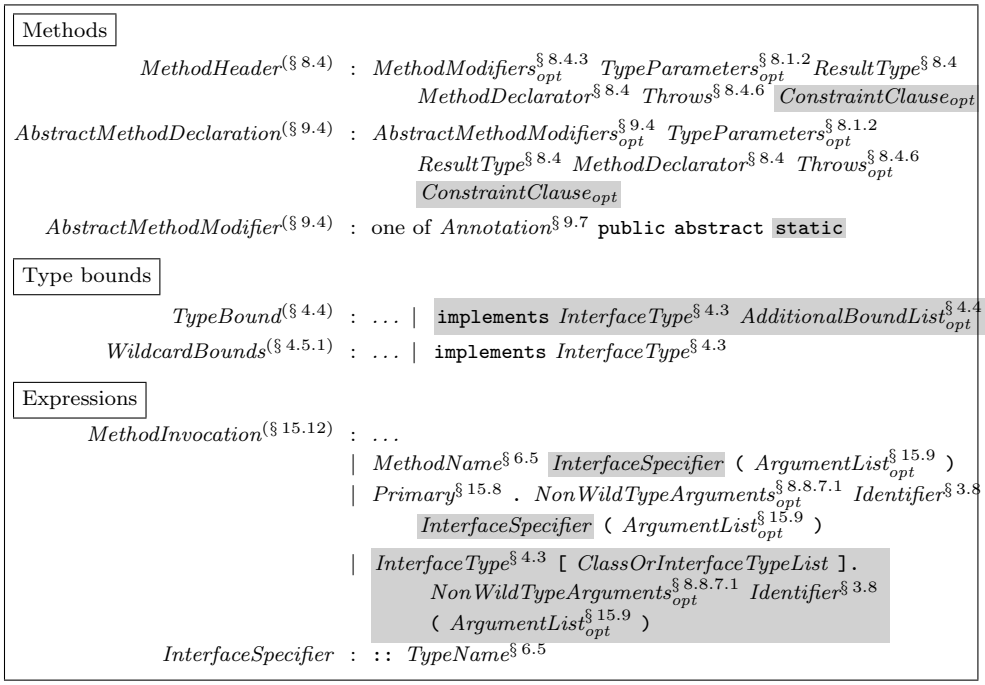

Figure 31: Syntax of JavaGI (2/2).