Polyglot: An Extensible Compiler Framework for Java

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers

Cornell University
{nystrom,clarkson,andru}@cs.cornell.edu

Abstract. Polyglot is an extensible compiler framework that supports the easy creation of compilers for languages similar to Java, while avoiding code duplication. The Polyglot framework is useful for domain-specific languages, exploration of language design, and for simplified versions of Java for pedagogical use. We have used Polyglot to implement several major and minor modifications to Java; the cost of implementing language extensions scales well with the degree to which the language differs from Java. This paper focuses on the design choices in Polyglot that are important for making the framework usable and highly extensible. Polyglot source code is available.

1 Introduction

Domain-specific extension or modification of an existing programming language enables more concise, maintainable programs. However, programmers construct domain-specific language extensions infrequently because building and maintaining a compiler is onerous. Better technology is needed. This paper presents a methodology for the construction of extensible compilers and also an application of this methodology in our implementation of Polyglot, a compiler framework for creating extensions to Java [17].

Language extension or modification is useful for many reasons:

- **Security.** Systems that enforce security at the language level may find it useful to add security annotations or rule out unsafe language constructs.
- **Style.** Some language features or idioms may be deemed to violate good style but may not be easy to detect with simple lexical analysis.
- **Teaching.** Students may learn better using a language that does not expose them to difficult features (e.g., inner classes) or confusing error messages [10].
- **Language design.** Implementation helps validate programming language designs.
- **Optimization.** New passes may be added to implement optimizations not performed by the base compiler or not permitted by the base language specification.
- **Static checking.** A language might be extended to support annotations necessary for static verification of program correctness [24], more powerful static checking of program invariants [8, 12, 11], or heuristic methods [9].

We refer to the original unmodified language as the *base language*; we call the modified language a language *extension* even if it is not backwards compatible.

When developing a compiler for a language extension, it is clearly desirable to build upon an existing compiler for the base language. The simplest approach is to copy the source code of the base compiler and edit it in place. This may be fairly effective

if the base compiler is carefully written, but it duplicates code. Changes to the base compiler—perhaps to fix bugs—may then be difficult to apply to the extended compiler. Without considerable discipline, the code of the two compilers diverges, leading to duplication of effort.

Our approach is different: the Polyglot framework implements an extensible compiler for the base language Java 1.4. This framework, also written in Java, is by default simply a semantic checker for Java. However, a programmer implementing a language extension may extend the framework to define any necessary changes to compilation process, including the abstract syntax tree and semantic analysis.

An important goal for Polyglot is *scalable extensibility*: an extension should require programming effort proportional only to the magnitude of the difference between the extended and base languages. Adding new AST node types or compiler passes should require writing code whose size is proportional to the change. Language extensions often require uniformly adding new fields and methods to an AST node and its subclasses; we require that this uniform *mixin* extension be implementable without subclassing all the extended node classes. Scalable extensibility is a challenge because it is difficult to simultaneously extend both types and the procedures that manipulate them [35, 43]. Existing programming methodologies such as *visitors* [16] improve scalable extensibility but are not a complete solution. In this paper we present a methodology that supports extension of both compiler passes and AST nodes and that addresses the mixin problem. The methodology uses abstract factories and delegation [16] to permit greater extensibility and code reuse than in previous extensible compiler designs. To our knowledge, Polyglot is the first compiler framework to provide scalable extensibility, including a solution to the mixin problem.

Polyglot has been used to implement more than a dozen Java language extensions of varying complexity. Our experience using Polyglot suggests that it is a useful framework for developing compilers for new Java-like languages. Some of the complex extensions implemented are Jif [27, 31], which extends Java with security types that regulate information flow, PolyJ [28, 29], which adds bounded parametric polymorphism to Java; and JMatch [30, 26], which extends Java with pattern matching and iteration features. Compilers built using Polyglot are themselves extensible; complex extensions such as Jif and PolyJ have themselves been extended. The framework is not difficult to learn: users have been able to build interesting extensions to Java within a day of starting to use Polyglot. The Polyglot source code is available.[1]

The rest of the paper is structured as follows. Section 2 gives an overview of the Polyglot compiler. Section 3 describes in detail our methodology for achieving scalable extensibility. Other Polyglot features that make writing an extensible compiler convenient are described in Section 4. Our experience using the Polyglot system to build various languages is reported in Section 5. Related work on extensible compilers and macro systems is discussed in Section 6, and we conclude in Section 7.

[1] At `http://www.cs.cornell.edu/Projects/polyglot`

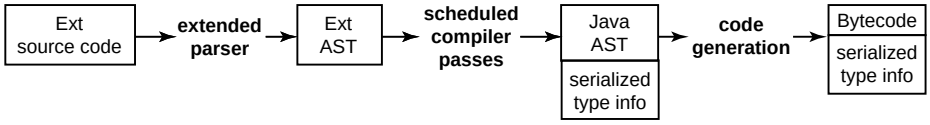

Fig. 1. Polyglot Architecture

2 Polyglot Overview

This section presents an overview of the various components of Polyglot and describes how they can be extended to implement a language extension. An example of a small extension is given to illustrate this process.

2.1 Architecture

A Polyglot extension is a source-to-source compiler that accepts a program written in a language extension and translates it to Java source code. It also may invoke a Java compiler such as `javac` to convert its output to bytecode.

The compilation process offers several opportunities for the language extension implementer to customize the behavior of the framework. This process, including the eventual compilation to Java bytecode [25], is shown in Fig. 1. In the figure, the name `Ext` stands for the particular extended language.

The first step in compilation is parsing input source code to produce an AST. Polyglot includes an extensible parser generator, PPG, that allows the implementer to define the syntax of the language extension as a set of changes to the base grammar for Java. PPG provides *grammar inheritance* [38, 34], which can be used to add, modify, or remove productions and symbols of the base grammar. PPG is implemented as a preprocessor for the CUP LALR parser generator [19].

The extended AST may contain new kinds of nodes either to represent syntax added to the base language or to record new information in the AST. These new node types are added by implementing the `Node` interface and optionally subclassing from an existing node implementation.

The core of the compilation process is a series of compilation passes applied to the abstract syntax tree. Both semantic analysis and translation to Java may comprise several such passes. The *pass scheduler* selects passes to run over the AST of a single source file, in an order defined by the extension, ensuring dependencies between source files are not violated. Each compilation pass, if successful, rewrites the AST, producing a new AST that is the input to the next pass. Some analysis passes (e.g., type checking) may halt compilation and report errors instead of rewriting the AST. A language extension may modify the base language pass schedule by adding, replacing, reordering, or removing compiler passes. The rewriting process is entirely functional; compilation passes do not destructively modify the AST. More details on our methodology are described in Section 3.

Compilation passes do their work using objects that define important characteristics of the source and target languages. A *type system* object acts as a factory for objects

```
1  tracked(F) class FileReader {
2    FileReader(File f) [] -> [F] throws IOException[] { ... }
3    int read() [F] -> [F] throws IOException[F] { ... }
4    void close() [F] -> [] { ... ; free this; }
5  }
```

Fig. 2. Example Coffer `FileReader`

representing types and related constructs such as method signatures. The type system object also provides some type checking functionality. A *node factory* constructs AST nodes for its extension. In extensions that rely on an intermediate language, multiple type systems and node factories may be used during compilation.

After all the compilation passes complete, the usual result is a Java AST. A Java compiler such as `javac` is invoked to compile the Java code to bytecode. The bytecode may contain serialized extension-specific type information used to enable separate compilation; we discuss separate compilation in more detail in Section 4.2.

2.2 An Example: Coffer

To motivate our design, we show a simple extension of Java that supports some of the resource management facilities of the Vault language [7]. This language, called Coffer, is a challenge for extensible compilers because it makes substantial changes to both the syntax and semantics of Java requiring identical modifications to many AST node types. Coffer allows a linear capability, or *key*, to be associated with an object. Methods of the object may be invoked only when the key is held. A key is allocated when its object is created and deallocated by a `free` statement in a method of the object. The Coffer type system regulates allocation and freeing of keys to guarantee statically that keys are always deallocated.

Fig. 2 shows a small Coffer program declaring a `FileReader` class that guarantees the program cannot read from a closed reader. The annotation `tracked(F)` on line 1 associates a key named `F` with instances of `FileReader`. Pre- and post-conditions on method and constructor signatures, written in brackets, specify how the set of held keys changes through an invocation. For example on line 2, the precondition `[]` indicates that no key need be held to invoke the constructor, and the postcondition `[F]` specifies that `F` is held when the constructor returns normally. The `close` method (line 4) frees the key; no subsequent method that requires `F` can be invoked.

The Coffer extension is used as an example throughout the next section. It is implemented by adding new compiler passes for computing and checking held key sets at each program point. Coffer's `free` statements and additional type annotations are implemented by adding new AST nodes and extending existing nodes and passes.

3 A Methodology for Scalable Extensibility

Our goal is a mechanism that supports scalable extension of both the syntax and semantics of the base language. The programmer effort required to add or extend a pass

should be proportional to the number of AST nodes non-trivially affected by that pass; the effort required to add or extend a node should be proportional to the number of passes the node must implement in an interesting way.

When extending or overriding the behavior of existing AST nodes, it is often necessary to extend a node class that has more than one subclass. For instance, the Coffer extension adds identical pre- and post-condition syntax to both methods and constructors; to avoid code duplication, these annotations should be added to the common base class of method and constructor nodes. The programmer effort to make such changes should be constant, irrespective of the number of subclasses of the node class. Inheritance is the appropriate mechanism when adding a new field or method to a single class. However, adding the same field or method to many different node types can quickly become tedious. This is true even in languages with multiple inheritance: a new subclass must be created for every node affected by the change. Modifying these subclasses later may require making identical changes to numerous classes. Solving this *mixin problem* is a key goal of our methodology.

Compilers written in object-oriented languages typically implement compiler passes using the *Visitor* design pattern [16]. However, visitors present several problems for scalable extensibility. In a non-extensible compiler, the set of AST nodes is usually fixed. The Visitor pattern permits scalable addition of new passes, but sacrifices scalable addition of AST node types. To allow specialization of visitor behavior for both the AST node type and the visitor itself, each visitor class implements a separate callback method for every node type. Thus, adding a new kind of AST node requires modifying *all* existing visitors to insert a callback method for the node. Visitors written without knowledge of the new node cannot be used with the new node because they do not implement the callback. The Visitor pattern also does not address the mixin problem; it was not intended to provide a mechanism for orthogonally extending node classes. A separate mechanism is needed to address this problem.

An alternative to the Visitor pattern is for each AST node class to implement a method for each compiler pass. However, this technique suffers from the dual problem: adding a new pass requires adding a method to all existing node types.

The remainder of this section presents a mechanism that achieves the goal of scalable extensibility. We first describe our approach to solving the mixin problem. We then show how our solution also addresses the other aspects of scalable extensibility.

3.1 Node Extension Objects and Delegates

We implement passes as methods associated with AST node objects; however, to achieve scalable extensibility, we introduce a delegation mechanism, illustrated in Fig. 3, that enables orthogonal extension and method override of nodes.

Since subclassing of node classes does not adequately address orthogonal extension of methods in classes with multiple subclasses, we add to each node object a field, labeled `ext` in Fig. 3, that points to a (possibly `null`) *node extension object*. The extension object (`CofferExt` in the figure) provides implementations of new methods and fields, thus extending the node interface without subclassing. These members are accessed by following the `ext` pointer and casting to the extension object type. In the example, `CofferExt` extends `Node` with `keyFlow()` and `checkKeys()` methods. Each

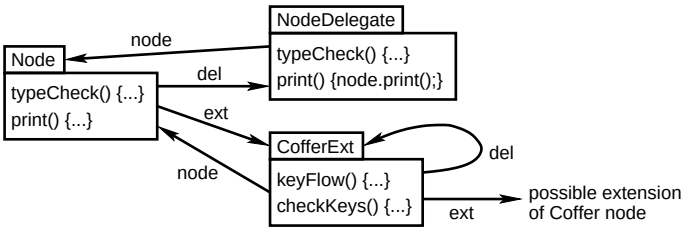

Fig. 3. Delegates and extensions

AST node class to be extended with a given implementation of these members uses the same extension object class. Thus, several node classes can be orthogonally extended with a single implementation, avoiding code duplication. Since language extensions can themselves be extended, each extension object has an `ext` field similar to the one located in the node object. In effect, a node and its extension object together can be considered a single node.

Extension objects alone, however, do not adequately handle method override when the base language is extended multiple times. The problem is that any one of a node's extension objects can implement the overridden method; a mechanism is needed to invoke the correct implementation. A possible solution to this problem is to introduce a *delegate* object for each method in the node interface. For each method, a field in the node points to an object implementing that method. Calls to the method are made through its delegate object; language extensions can override the method simply by replacing the delegate. The delegate may implement the method itself or may invoke methods in the node or in the node's extension objects.

Because maintaining one object per method is cumbersome, the solution used in Polyglot is to combine delegate objects and to introduce a single delegate field for each node object—illustrated by the `del` field in Fig. 3. This field points to an object implementing the entire `Node` interface, by default the node itself. To override a method, a language extension writer creates a new delegate object containing the new implementation or code to dispatch to the new implementation. The delegate implements `Node`'s other methods by dispatching back to the node. Extension objects also contain a `del` field used to override methods declared in the extension object interface.

Calls to all node methods are made through the `del` pointer. thus ensuring that the correct implementation of the method is invoked if the delegate object is replaced by a language extension. Thus, in our example, the node's `typeCheck` method is invoked via `n.del.typeCheck();` the Coffer `checkKeys` method is invoked by following the node's `ext` pointer and invoking through the extension object's delegate: `((CofferExt) n.ext).del.checkKeys()`. An extension of Coffer could replace the extension object's delegate to override methods declared in the extension, or it could replace the node's delegate to override methods of the node. To access Coffer's type-checking functionality, this new node delegate may be a subclass of Coffer's node delegate class or may contain a pointer to the old delegate object.

3.2 AST Rewriters

Most passes in Polyglot are structured as functional AST rewriting passes. Factoring out AST traversal code eliminates the need to duplicate this code when implementing new passes. Each pass implements an *AST rewriter* object to traverse the AST and invoke the pass's method at each node. At each node, the rewriter invokes a `visitChildren` method to recursively rewrite the node's children using the rewriter and to reconstruct the node if any of the children are modified. A key implementation detail is that, when a node is reconstructed, the node is *cloned* and the clone is returned. Cloning ensures that class members added by language extensions are correctly copied into the new node. The node's delegates and extensions are cloned with the node.

Each rewriter implements `enter` and `leave` methods, both of which take a node as argument. The `enter` method is invoked before the rewriter recurses on the node's children using `visitChildren` and may return a new rewriter to be used for rewriting the children. This provides a convenient means for maintaining symbol table information as the rewriter crosses lexical scopes; the programmer need not write code to explicitly manage the stack of scopes, eliminating a potential source of errors. The `leave` method is called after visiting the children and returns the rewritten AST rooted at the node.

3.3 Scalable Extensibility

A language extension may extend the interface of an AST node class through an extension object interface. For each new pass, a method is added to the extension object interface and a rewriter class is created to invoke the method at each node. For most nodes, a single extension object class is implemented to define the default behavior of the pass, typically just an identity transformation on the AST node. This class is overridden for individual nodes where non-trivial work is performed for the pass.

To change the behavior of an existing pass at a given node, the programmer creates a new delegate class implementing the new behavior and associates the delegate with the node at construction time. Like extension classes, the same delegate class may be used for several different AST node classes, thus allowing functionality to be added to node classes at arbitrary points in the class hierarchy without code duplication.

New kinds of nodes are defined by new node classes; existing node types are extended by adding an extension object to instances of the class. A factory method for the new node type is added to the node factory to construct the node and, if necessary, its delegate and extension objects. The new node inherits default implementations of all compiler passes from its base class and from the extension's base class. The new node may provide new implementations using method override, possibly via delegation. Methods need be overridden only for those passes that need to perform non-trivial work for that node type.

Fig. 4 shows a portion of the code implementing the Coffer key-checking pass, which checks the set of keys held when control enters a node. The code has been simplified in the interests of space and clarity. At each node in the AST, the pass invokes through the `del` pointer the `checkKeys` method in the Coffer extension, passing in the set of held keys (computed by a previous data-flow analysis pass). Since most AST nodes are not affected by the key-checking pass, a default `checkKeys` method

```
class KeyChecker extends Rewriter {
  Node leave(Node n) {
    ((CofferExt) n.ext).del.checkKeys(held_keys(n));
    return n;
  }
}

class CofferExt {
  Node node; CofferExt del;
  void checkKeys(Set held_keys) { /* empty */ }
}

class ProcedureCallExt extends CofferExt {
  void checkKeys(Set held_keys) {
    ProcedureCall c = (ProcedureCall) node;
    CofferProcedureType p = (CofferProcedureType) c.callee();
    if (! held_keys.containsAll(p.entryKeys()))
      error(p.entryKeys() + " not held at " + c);
  }
}
```

Fig. 4. Coffer key checking

implemented in the base `CofferExt` class is used for these nodes. For other nodes, a non-trivial implementation of key checking is required.

Fig. 4 also contains an extension class used to compute the held keys for method and constructor calls. `ProcedureCall` is an interface implemented by the classes for three AST nodes that invoke either methods or constructors: method calls, `new` expressions, and explicit constructor calls (e.g., `super()`). All three nodes implement the `checkKeys` method identically. By using an extension object, we need write this code only once.

4 Other Implementation Details

In this section we consider some aspects of the implementation of Polyglot, not directly related to scalable extensibility.

4.1 Data-Flow Analysis

Polyglot provides an extensible data-flow analysis framework. In Java implementation, this framework is used to check the that variables are initialized before use and that all statements are reachable; extensions perform additional data-flow analyses to enable optimizations or other transformations. Polyglot provides a rewriter in the base compiler framework that constructs the control-flow graph of the program. Intraprocedural data-flow analyses can then be performed on this graph by implementing the meet and transfer functions for the analysis.

One well-known difficulty in constructing the control-flow graph for Java programs are `finally` blocks [15]. The `finally` block is essentially a local subroutine that is

invoked for every exit from the associated `try` block. To accurately capture the control flow of Java methods, we create a copy of the flow graph for the `finally` block for each possible entry into the block.

4.2 Separate Compilation

Java compilers use type information stored in Java class files to support separate compilation. For many extensions, the standard Java type information in the class file is insufficient. Polyglot injects type information into class files that can be read by later invocations of the compiler to provide separate compilation. No code need be written for a language extension to use this functionality for its extended types. Before performing Java code generation, Polyglot uses the Java serialization facility to encode the type information for a given class into a string, which is then compressed and inserted as a final static field into the AST for the class being serialized. When compiling a class, the first time a reference to another class is encountered, Polyglot loads the class file for the referenced class and extracts the serialized type information. The type information is decoded and may be immediately used by the extension.

Serialization of a type object o recursively follows all non-transient pointers from o, serializing the object graph rooted at o. References to type objects created for code in the AST of other source files are *not* followed because these objects are serialized into their respective ASTs. Instead, such references are replaced with a placeholder object; when the placeholder is deserialized, the referenced class is loaded into the compiler if it is not already present.

4.3 Quasiquoting

To generate Java output, language extensions translate their ASTs to Java ASTs and rely on the code generator of the base compiler to output Java code. To enable AST rewriting we have used PPG to extend Polyglot's Java parser with the ability to generate an AST from a string of Java code and a collection of AST nodes to substitute into the generated AST. This feature provides many of the benefits of quasiquoting in Lisp or Scheme [36, 21].

Our quasiquoting facility is implemented by extending the base Java grammar with new terms to be used as placeholders for expressions, statements, types, or class members. The parser substitutes AST nodes passed in by the caller for the placeholders much as the C `printf` function inserts its arguments into the output stream as the format string is parsed. Applying a similar PPG specification to a language extension's grammar allows ASTs to be generated for the extension language; thus, this facility can be used to generate ASTs for intermediate code.

Using our technique, syntax errors in the source string may not be caught until after the compiler has been deployed. True quasiquoting, as found in languages such as Scheme [21], would detect syntax errors when the extension compiler is itself compiled. Quasiquoting would thus be a useful extension to Java in which to implement Polyglot.

5 Experience

More than a dozen extensions of varying sizes have been implemented using Polyglot, for example:

- Jif is a Java extension that provides information flow control and features to ensure the confidentiality and integrity of data [27, 31].
- Jif/split is an extension to Jif that partitions programs across multiple hosts based on their security requirements [42].
- PolyJ is a Java extension that supports bounded parametric polymorphism [28, 29].
- JMatch is a Java extension that supports pattern matching and logic programming features [30].
- Coffer, as previously described, adds resource management facilities to Java.
- PAO ("primitives as objects") allows primitive values to be used transparently as objects via automatic boxing and unboxing,
- A covariant return extension restores the subtyping rules of Java 1.0 Beta [37] in which the return type of a method could be covariant in subclasses. The language was changed in the final version of Java 1.0 [17] to require the invariance of return types.

The major extensions add new syntax and make substantial changes to the language semantics. We describe the changes for Jif and PolyJ in more detail below. The smaller extensions, such as support for covariant return types, require more localized changes.

5.1 Jif

Jif is an extension to Java that permits static checking of information flow policies. In Jif, the type of a variable may be annotated with a *label* specifying a set of principals who own the data and a set of principals that are permitted to read the data. Labels are checked by the compiler to ensure that the information flow policies are not violated.

The base Polyglot parser is extended using PPG to recognize security annotations and new statement forms. New AST node classes are added for labels and for new statement and expression forms concerning security checks. The new AST nodes and nearly all existing AST nodes are also extended with security context annotations. These new fields are added to a Jif extension class. To implement information flow checking, a `labelCheck` method is declared in the Jif extension object. Many nodes do no work for this pass and therefore can inherit a default implementation declared in the base Jif extension class. Extension objects installed for expression and statement nodes override the `labelCheck` method to implement the security typing judgment for the node. Delegates were used to override type checking of some AST nodes to disallow static fields and inner classes since they may provide an avenue for information leaks.

Following label checking, the Jif AST is translated to a Java AST, largely by erasing security annotations. The new statement and expression forms are rewritten to Java syntax using the quasiquoting facility discussed in Section 4.3.

Jif/split further extends Jif to partition programs across multiple hosts based on their security requirements. The syntax of Jif is modified slightly to also support integrity annotations. New passes, implemented in extension objects, partition the Jif/split program into several Jif programs, each of which will run on a separate host.

Extension	Token count	Percent of Base Polyglot
base Polyglot	160036	100%
empty	1152	<1%
parameterized classes	3233	2%
Jif	122541	77%
JMatch	104840	66%
PolyJ	78159	49%
Coffer	21731	14%
PAO	3788	2%
covariant return	1572	1%

Table 1. Extension size

5.2 PolyJ

PolyJ is an extension to Java that supports parametric polymorphism [28, 29]. Classes and interfaces may be declared with zero or more parameters constrained by *where clauses*. The base Java parser is extended using PPG, and AST node classes are added for where clauses and for new type syntax. Further, the AST node for class declarations is extended via inheritance to allow for type parameters and where clauses.

The PolyJ type system customizes the behavior of the base Java type system and introduces judgments for parameterized and instantiated types. A new pass is introduced to check that the types on which a parameterized class is instantiated satisfy the constraints for that parameter, as described in [28].

The base compiler code generator is extended to generate code not only for each PolyJ source class, but also for *adapter classes*, which "adapt" a parameterized type, such as `Map[K,V]`, to a particular instantiation of that type, such as `Map[String, int]`.

5.3 Results

The size of the extensions discussed in this paper are shown in Table 1. Sizes are given in number of tokens for source files, including Java, CUP, and PPG files.

These results demonstrate that the cost of implementing language extensions scales well with the degree to which the extension differs from its base language. Simple extensions such as the covariant return extension that differ from Java in small, localized ways can be implemented by writing only small amounts of code. To measure the overhead of simply creating a language extension, we implemented an empty extension that makes no changes to the Java language; the overhead includes empty subclasses of the base compiler node factory and type system classes, an empty PPG parser specification, and code for allocating these subclasses.

PolyJ, which has large changes to the type system and to code generation, requires implementing only about half the amount of code as the base Java compiler. For historical reasons, PolyJ generates code by overriding the Polyglot code generator to directly output Java. The size of this code could be reduced by using quasiquoting. Jif requires

a large amount of extension code because label checking in Jif is more complex than the Java type checking that it extends. Both PolyJ and Jif extend a small "abstract extension" of Java that provides some support for parameterized classes. Much of the JMatch overhead is accounted for by extensive changes to add complex statement and expression translations.

As a point of comparison, the base Polyglot compiler (which implements Java 1.4) and the Java 1.1 compiler, `javac`, are nearly the same size when measured in tokens. Thus, the base Polyglot compiler implementation is reasonably efficient. To be fair to `javac`, we did not count its code for bytecode generation. About 10% of the base Polyglot compiler consists of interfaces used to separate the interface hierarchy from the class hierarchy. The `javac` compiler is not implemented this way.

Implementing small extensions has proved to be fairly easy. We asked a programmer previously unfamiliar with the framework to implement the covariant return type extension; this took one day. The same programmer implemented several other small extensions within a few days.

5.4 Discussion

In implementing Polyglot we found, not surprisingly, that application of good object-oriented design principles greatly enhances Polyglot's extensibility. Rigorous separation of interfaces and classes permit implementations to be more easily extended and replaced; calls through interfaces ensure the framework is not bound to any particular implementation of an interface. The Polyglot framework almost exclusively uses *factory methods* to create objects [16] because the code that creates objects may instantiate any class with the appropriate type. An extension is free to change the implementation of objects by overriding factory methods.

We chose to implement Polyglot using only standard Java features, but it is clear that several language extensions—some of which we have implemented using Polyglot—would have made it easier to implement Polyglot. Multimethods (e.g., [5]) would have simplified the dispatching mechanism needed for our methodology. Open classes [6] would probably have provided a cleaner solution to the extensibility problem, particularly in conjunction with multimethods. Multiple inheritance and mixins (e.g., [3, 13]) in particular would facilitate application of an extension to many AST nodes at once. Built-in quasiquoting support would make translation more efficient, though the need to support several target languages would introduce some difficulties. Covariant modification of method return types would eliminate many unnecessary type casts, as would parametric polymorphism [28, 32].

6 Related Work

We present work that is closely related to Polyglot, including other extensible compilers, macro systems, and visitor patterns. No work of which we are aware presents a solution to the mixin problem in a language with single inheritance.

JaCo is an extensible compiler for Java written in an extended version of Java [44] that supports ML-style pattern matching. However, it relies on a new language feature—extensible algebraic datatypes [43]—to address the difficulty of handling new data types

without changing existing code. Polyglot achieves scalable extensibility while relying only on features available in Java.

The Jakarta Tools Suite (JTS) [2] is a pair of tools for implementing Java preprocessors to create domain-specific languages. Extensions of a base language are encapsulated as components that define the syntax and semantics of the extension. A fundamental difference between JTS and Polyglot is that JTS is not concerned with much beyond the syntactic analysis of the host or extension language [2, section 4]. This makes JTS more like a macro system in which the macros are defined by extending the compiler rather than declaring them in the source code.

Macro systems and preprocessors are generally concerned only with syntactic extensions to a language. Recent systems for use in Java include EPP [20], JSE [14], and JPP [22]. Maya [1] is a generalization of macro systems that uses generic functions and multimethods to allow extension of Java syntax. Semantic actions can be defined as multimethods on those generic functions. It is not clear how these systems scale to support semantic checking for large extensions to the base language.

OpenJava [39] uses a meta-object protocol (MOP) to allow manipulation of a program's structure. The MOP used is much like a superset of that available through Java's reflection API, allowing access to classes, methods and fields, as well as statements and expressions. OpenJava allows very limited extension of syntax, but through its MOP exposes much of the semantic structure of the program.

The original Visitor design pattern [16] has led to many refinements (e.g., [33, 4, 40]). Extensible Visitors [23] and Staggered Visitors [41] both enhance the extensibility of the visitor pattern to facilitate adding new node types, but neither these nor the other refinements mentioned above consider the mixin problem. Staggered Visitors rely on multiple inheritance to extend visitors with support for new nodes.

7 Conclusions

Our original motivation for developing the Polyglot compiler framework was simply to provide a publicly available Java front end that could be easily extended to support new languages. We discovered that the existing approaches to extensible compiler construction within Java did not solve to our satisfaction the problem of scalable extensibility including mixins. Our extended visitor methodology is simple, yet improves on the previous solutions to the extensibility problem. Other Polyglot features such as extensible parsing, pass scheduling, quasiquoting, and type signature insertion are also quite useful.

Our experience using Polyglot to build a number of substantial compilers has shown that it is an effective way to produce compilers for Java-like languages. We have used the framework for several significant language extensions that modify Java syntax and semantics in complex ways. We hope that the public release of this software in source code form will facilitate experimentation with new features for object-oriented languages.

References

1. Jason Baker and Wilson C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proc. of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, Berlin, Germany, June 2002.
2. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53, Victoria, BC, Canada, 1998. IEEE.
3. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
4. Martin Bravenboer and Eelco Visser. Guiding visitors: Separating navigation from computation. Technical report, Institute of Information and Computing Sciences, University of Utrecht, November 2001.
5. Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.
6. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.
7. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.
8. David Detlefs. An overview of the extended static checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM (SIGSOFT), January 1996.
9. Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Fourth Usenix Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.
10. Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, 1997.
11. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proc. of the ACM SIGPLAN '00 Conference on Program Language Design and Implementation (PLDI)*, pages 219–232, Vancouver, Canada, June 2000.
12. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, Berlin, Germany, June 2002.
13. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
14. D. K. Frayne and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the 2001 Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '01)*, pages 31–42, Tampa, FL, USA, 2001.
15. Stephen N. Freund. The costs and benefits of Java bytecode subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA*, October 1998.

16. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.
17. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.
18. Carl Gunter and John C. Mitchell, editors. *Theoretical aspects of object-oriented programming*. MIT Press, 1994.
19. Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew Appel. CUP LALR parser generator for Java, 1996. Software release. http://www.cs.princeton.edu/-∼appel/modern/java/CUP/.
20. Yuuji Ichisugi and Yves Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *Proc. ISCOPE '97*, LNCS 1343, pages 153–160. Springer, 1997.
21. Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, October 1998. Available at http://www.schemers.org/Documents/Standards/R5RS.
22. Joseph R. Kiniry and Elaine Cheong. JPP: A Java pre-processor. Technical Report CS-TR-98-15, California Institute of Technology, Pasadena, CA, September 1998.
23. Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proc. ECOOP '98*, pages 91–113, 1998.
24. Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, Minneapolis, Minnesota, 2000.
25. T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.
26. Jed Liu and Andrew C. Myers. JMatch: Java plus pattern matching. Technical Report TR2002-1878, Computer Science Department, Cornell University, October 2002. Software release at `http://www.cs.cornell.edu/projects/jmatch`.
27. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.
28. Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.
29. Andrew C. Myers, Barbara Liskov, and Nicholas Mathewson. PolyJ: Parameterized types for Java. Software release. Located at http://www.cs.cornell.edu/polyj, July 1998–present.
30. Andrew C. Myers and Jed Liu. JMatch: Abstract iterable pattern matching for Java. In *Proc. 5th Int'l Symp. on Practical Aspects of Declarative Languages*, New Orleans, LA, January 2003. To appear.
31. Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.
32. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.
33. Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, 1998.
34. Terence Parr and Russell Quong. ANTLR: A predicated-LL(k) parser generator. *Journal of Software Practice and Experience*, 25(7), 1995.
35. John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorith-*

mic Languages, pages 157–168. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975. Reprinted in [18], pages 13–23.
36. Guy Steele. *Common LISP: the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.
37. Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at ftp://ftp.javasoft.com/docs/javaspec.ps.zip.
38. Reasoning Systems. Dialect user's guide, 1990.
39. Michiaki Tatsubori, Shigeru Chiba, Marc-Oliver Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag, July 2000.
40. Joost Visser. Visitor combination and traversal control. In *Proceedings of the 2001 Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '01)*, pages 270–282, Tampa, FL, USA, October 2001.
41. J. Vlissides. Visitors in frameworks. *C++ Report*, 11(10), November 1999.
42. Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, pages 1–14, Banff, Canada, October 2001.
43. Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc. 6th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Firenze, Italy, September 2001.
44. Matthias Zenger and Martin Odersky. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.