Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions*

Eric Van Wyk
University of Minnesota
evw@cs.umn.edu

Derek Bodin
University of Minnesota
bodin@cs.umn.edu

Paul Huntington
University of Minnesota
johnspa@cs.umn.edu

ABSTRACT

We show how new syntactic forms and static analysis can be added to a programming language to support abstractions provided by libraries. Libraries have the important characteristic that programmers can use multiple libraries in a single program. Thus, any attempt to extend a language's syntax and analysis should be done in a composable manner so that similar extensions that support other libraries can be used by the programmer in the same program. To accomplish this we have developed an extensible attribute grammar specification of Java 1.4 written in the attribute grammar specification language Silver. Library writers can specify, as an attribute grammar, new syntax and analysis that extends the language and supports their library. The Silver tools automatically compose the grammars defining the language and the programmer-selected language extensions (for their chosen libraries) into a specification for a new custom language that has language-level support for the libraries. We demonstrate how syntax and analysis are added to a language by extending Java with syntax from the query language SQL and static analysis of these constructs so that syntax and type errors in SQL queries can be detected at compile-time.

1. INTRODUCTION

Libraries play a critical role in nearly all modern programming languages. The Java libraries, C# libraries, the C++ Standard Template Library, and the Haskell Prelude all provide important abstractions and functionality to programmers in those language; learning a programming language now involves learning the intricacies of its libraries as well. The libraries are as much a part of these languages as their type systems. Using libraries to define new abstractions for a language helps to keep the definition of the language simpler than if these features where implemented as first class constructs of the language.

*Different aspects of this work are partially funded by NSF CAREER Award #0347860 and the McKnight Foundation.

LCSD '06 Portland, Oregon USA

An important characteristic of libraries is their compositionality. A programmer can use multiple libraries, from different sources, in the same application. Thus, libraries that support specific domains can be used in applications with aspects that cross multiple domains. For example, a Java application that stores data in a relational database, processes the data and displays it using a graphical user interface may use both the JDBC and the Swing libraries. Furthermore, abstractions useful to much smaller communities, such as the computational geometry abstractions in the CGAL C++ library, can also be packaged as libraries.

Libraries have a number of drawbacks, however. As mechanisms for extending languages they provide no means for library writers to add new syntax that may provide a more readable means of using the abstraction in a program. Traditional libraries provide no effective means for library writers to specify any static semantic analysis that the compiler can use to ensure that the library abstractions (methods or functions) are used correctly by the programmer. When libraries embed domain specific languages into the "host" language, as the JDBC library embeds SQL into Java, there is no means for statically checking that expressions in the embedded language are free of syntax and type errors. This is a serious problem with the JDBC library since syntax and type errors are not discovered at compile time but at run time. Traditional libraries also provide no means for specifying optimizations of method and function calls.

These drawbacks, especially in libraries for database access, have led some to implement the abstractions not as libraries but as constructs and types in the language. There is an trend in database systems towards more tightly integrating the application program with the database queries. Jim Gray [10] calls this removing the "inside the database" and "outside the database" dichotomy. In many cases, this means more tightly integrating the Java application program with the SQL queries to be performed on a database server. SQLJ is an example of this. Part 0 of the SQLJ standard [7] specifies how static database queries can be written directly in a Java application program. An SQLJ compiler checks these queries for syntax and type errors. This provides a much more natural programming experience than that provided by a low level API such as JDBC (Java DataBase Connector) which require the programmer to treat database query commands as Java Strings that are passed, as strings, to a database server where they are not checked for syntactic or type correctness until run time. More re-

cently, $C\omega$ [3] and the Microsoft LINQ project [15] have extended C# and the .Net framework to directly support the querying of relational data.

These extended languages have added relational data query constructs because the technologies have matured to a relatively stable point and because very many programs are written that can make use of these features. Thus, if one is working in this domain, one can benefit from a language that directly supports the task at hand. Programmers working in less popular domains, however, are left with the library approach as it is the only way in which their domain-specific abstractions can be used in their programs. In the approach of SQLJ, $C\omega$, and LINQ, a new monolithic language with new features is created, but there is no way for other communities to further extend Java or C# with new syntax and semantic analysis to support their domains.

In this paper we present a different, more general, approach to integrating programming and database query languages based on extensible languages and illustrate how new syntax and static analysis can be added to library-based implementations of new abstractions. The key characteristic of this approach is that multiple language extensions can be composed to form a new extended language that supports all aspects of a programming task. We have developed several modular, composable, language extensions to Java. In this paper we describe the extension that embeds SQL into Java to provide syntax and type checking for SQL queries and thus supports the implementation of these features in the JDBC library. We have built other extensions with domain-specific language features; one specifies program transformations that simplify the writing of robust and efficient computational geometry programs. Another general purpose extension adds pattern matching constructs from Pizza [17] to Java. Java and the language extensions are all specified as attribute grammars written in the attribute grammar specification language Silver. The Silver tools can automatically compose the grammars defining the host language Java and a programmer selected set of extensions to create a specification of a custom extended version of Java that has the features relevant to different specific domains. The tools then translate the specification to an executable compiler for the language.

Section 2 introduces the extensible language framework and its supporting tools. Section 3 describes a modular SQL extension to Java that we have constructed in order to illustrate what is possible in the framework. Section 4 provides the specifications of a subset of Java (Section 4.1) and some of the extension constructs (Section 4.2) to illustrate how the full extension in Section 3 was implemented. Section 5 describes related work, future work, and concludes.

2. EXTENSIBLE LANGUAGE SPECIFICATIONS AND SUPPORTING TOOLS

An extensible compiler allows the programmer to import the unique combination of general-purpose and domain-specific language features that raise the level of abstraction to that of a particular problem domain. These features may be new language constructs, semantic analyses, or optimizing program transformations, and are packaged as *modular language extensions*. Language extensions can be as simple as a *for-each* loop that iterates over collections or the set of SQL language constructs described in this paper.

To understand the type of language extensibility that we seek, an important distinction is made between two activities: (*i*) *implementing a language extension*, which is performed by a domain-expert feature designer and (*ii*) *selecting the language extensions* that will be imported into an extensible language in order to create an extended language. This second activity is performed by a programmer. This is the same distinction seen between library writers and library users. This distinction and the way that extensible languages and language extensions are used in our framework is diagrammed in Figure 1.

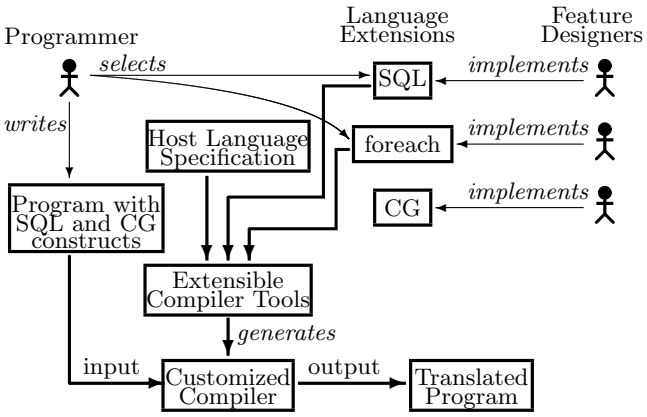

Figure 1: Using Extensible Languages and Language Extensions.

From the programmer's perspective, importing new language features should be as easy as importing a library in a traditional programming language. We want to maintain the compositional nature of libraries. They need only *select* the language extensions they want to use (perhaps the SQL and geometric (CG) extensions shown in Figure 1) and write their program to use the language constructs defined in the extensions and the "host" language. They need not know about the *implementation* of the host language or the extensions. The specifications for the selected language extensions and the host language are provided to the extensible compiler tools that generate a customized compiler. This compiler implements the unique combination of language features that the programmer needs to address the particular task at hand. Thus, there is an initial "compiler generation" step that the tools, at the direction of the programmer, must perform. Language extensions are not loaded into the compiler during compilation.

The feature designer's perspective is somewhat different; they are typically sophisticated domain experts with some knowledge of the implementation of the host language being extended. Critically, feature designers do not need to know about the implementations of other language extensions since they will not be aware of which language extensions a programmer will import. This paper shows how the functionality provided by a library can be enhanced by language extensions that provide new syntax to represent the abstractions provided by the library and new static analysis that can ensure that the library is used correctly.

2.1 Attribute grammars and Silver

Extensible languages and language extensions in our framework are based on *attribute grammars*. The extensible host language is specified as a complete attribute grammar and the language extensions are specified as attribute grammar fragments. These are written in Silver, an attribute grammar specification language developed to support this. The Silver extensible compiler tools combine the attribute grammar specifications of the host language and the programmer selected language extension to create an attribute grammar specification for the custom extended language desired by the programmer. An attribute grammar evaluator for this grammar implements the compiler for the extended language. We choose to define languages and extensions as attribute grammars, enhanced with with a mechanism called *forwarding* [19], because language specifications defined in the way can be automatically combined by the tools.

In Silver, attribute grammars are package as modules defining either a host language or a language extension. The module `edu:umn:cs:melt:java14` defines Java 1.4. It defines the concrete syntax of the language, the abstract syntax and the semantic analyses required to do most type checking analyses and to do package/type/expression name disambiguation. We continue to extend this with additional attribute definitions specifying additional static analyses. Our aim is to perform all static analyses performed by a traditional Java compiler. The grammar defines most aspects of a Java compiler but it does not specify byte-code generation. Language extensions add new constructs, like the SQL constructs described here, and their translation to pure Java 1.4 code, that a traditional Java compiler then converts to byte-codes for execution. The static analysis we perform is to support analysis of extensions and to ensure that any statically detectable errors (such as type errors and access violations) in the extended Java language are caught so that erroneous code is not generated. Programmers should not be expected to look at the generated Java code; errors should be reported on the code that they write.

In Section 4, we show Silver specifications for a simplified versions of our Java 1.4 grammar and the SQL grammar. We also show how the host language is composed with extensions to create an extended language specification.

3. A MODULAR LANGUAGE EXTENSION TO JAVA THAT EMBEDS SQL

In this section we describe a language extension that adds new constructs and semantic analyses that embed SQL into Java. The functionality is provided by the JDBC library — what is added is new syntax and analysis to statically detect syntax and type errors in SQL queries.

This extension specifies SQL productions for statically checking the syntax and type correctness of static SQL queries and queries which incorporate values from Java variables and expressions. It also allows for the dynamic creation of SQL queries in such a way that ensures that they are syntactically correct but it does not statically check the type correctness of dynamically generated queries. The integration component of the extension provides new constructs for registering database drivers, setting up connections, specifying and verifying the types of fields in tables on the database server that are used in the application, as well as executing queries and commands on the database server. The extended language will perform some semantic analysis on the SQL constructs to check for errors and then translate the constructs into pure Java code that uses the JDBC library. Figure 2 provides a code fragment written in the Java+SQL extended language. Figure 3 show the generated Java code fragment that the Java+SQL fragment translates to.

```
1.  register driver "com.mysql.jdbc.Driver" ;
2.  connection c="jdbc:mysql://db.domain.com/db..";
3.  import table person [ VARCHAR first_name ,
     VARCHAR last_name, INTEGER age ] ;
4.  ResultSet rs = using c query {
      SELECT last_name FROM person WHERE
        first_name LIKE "derek" };
5.  String s = "derek" ; int i = 4 ;
6.  rs = using c query { SELECT first_name
          FROM person WHERE first_name LIKE s };
7.  rs = using c query { SELECT first_name
          FROM person WHERE age > i } ;
8.  rs = using c query { SELECT first_name FROM person
          WHERE first_name LIKE $("der" + "ek") };
9.  sqlExpr e1=sql expr {person.first_name LIKE s};
10. sqlExpr e2=sql expr {e1 AND last_name LIKE "bodin"};
11. rs = using c query
     { SELECT first_name FROM person WHERE e2 };
```

Figure 2: SQL/Java example program fragment.

Static SQL queries: First consider lines 1-4 in Figure 2 that use constructs in the SQL extension to set up a connection to a database server and execute a static SQL query. Here three new statements added by the extension are used to register a database driver, to create a *connection* `c` to a specific database server, and to make the `person` table from that database available for use in the program. The `import table` construct specifies the fields and their type in the person database. At compile time, the declaration of the `person` table and the types of its fields are entered into an environment (symbol table) that is referenced during the type checking of the static queries.[1] Line 4 shows a statically specified SQL query.

In this example, since there are no syntax or type errors, the Java code in lines 1-4 of Figure 3 will be generated. The `register driver` and `connection` statement translate into the expected JDBC calls. The `import table` construct defines the type information used in the `using` and SQL constructs to perform static type checking, but it translates to the empty statement since it has no implementation in the generated Java code. The `using query` construct is translated to the expected JDBC `createStatement` and `executeQuery` methods calls on the JDBC Connection object `c`.

Accessing Java variables and expressions: The SQL

[1]Although not implemented in the current version of the extension, this construct could consult the database schema at compile time (as is done in SQLJ) to verify that the types in the database scheme match those given here.

```
1.  Class.forName ( "com.mysql.jdbc.Driver" ) ;
2.  Connection c = DriverManager.getConnection (
     "jdbc:mysql://db.domain.com:3306/db ...");
3.  // empty statement, was import
4.  ResultSet rs = c.createStatement().executeQuery(
     "SELECT " + "first_name" + " FROM " + "person" +
     "WHERE " + "first_name" + " LIKE " + "\"derek\" );
5.  String s = "derek" ;  int i = 4 ;
6.  rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
      + "WHERE " + "first_name" + " LIKE " + s);
7.  rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
      + "WHERE " + "age" + " > " + i);
8.  rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
      ++ "WHERE " + "first_name" + " LIKE " +
       ("der" + "ek"));
9.  String e1 = "person.first_name" + " LIKE " + s;
10. String e2 = e1 + " AND " + "last_name" +
                " LIKE " + "\"bodin\"" ;
11. rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
       + "WHERE " + e2) ;
```

Figure 3: Translation to pure Java of SQL/Java example program fragment.

queries defined in the SQL extension can access values from Java variables and expressions and still statically check that they are syntactically and type correct. For example, consider the lines 4-7 of Figure 2. The Java+SQL compiler will type-check these queries and, since no type errors exist, generate the pure Java code seen in lines 4-7 of Figure 3 (modulo reformatting). The examples translate to Java code that creates a string that contains the query using the value of variable `s` or `i`. The third example, computes the value of the expression whose value is included in the query. The type checking of these queries is possible here because the productions in the language extension have access to the attributes of the host language attribute grammar that contain the names and types of the in-scope variables. Because `s` is not a field in the table `person` but is an in-scope local variable the reference to `s` in line 6 of Figure 2 is to the local variable. In line 8, the `$(...)` notation is used to embed Java expressions in SQL expressions.

Dynamic creation of SQL queries: The SQL extension also allows for the creation of dynamic SQL queries that can be statically verified to be syntactically correct. Consider lines 9–11 of Figure 2 that makes use of the new type `sqlExpr`. The variables `e1` and `e2` of type `sqlExpr` hold syntactically correct SQL expressions. SQL expressions are written according to the SQL grammar but are wrapped in a `sql expr` { ... } construct to avoid conflicts and ambiguities in the parser that may arise since the expression language of SQL and the expression language of Java are similar. The Java code generated for the SQL+Java code is show in lines 9–11 of Figure 3.

In the generated Java code of Figure 3, the queries are syntactically correct but the type information that is present in the extension constructs of Figure 2 is gone and queries are represented as strings. Thus, statically checking the syntactic correctness of queries constructed as strings requires a much more sophisticated analysis, like that found in the Java String Analyzer [5] and incorporated in the JDBC Checker tool [9]. The JDBC checker does check that dynamically created queries are also type correct. That is something that this simple extension does not do. Nothing prohibits that kind of sophisticated analysis being done by this tool however. In fact, extensible languages provide the right "hooks" for extracting information used by such analysis. The control flow information needed by JSA can be generated by appropriate attribute definitions.

Some may justifiably question some of the design decisions of this extension. For example, should we recognize `s` in line 6 of Figure 2 as a local variable, or should we require the use of the `$(...)` notation. It is not our aim to answer these sorts of questions. Our goal is to show how syntax and static analyses can be added to a programming language, not to argue the merits of specific SQL extension constructs.

4. EXTENSIBLE LANGUAGE IMPLEMENTATIONS VIA ATTRIBUTE GRAMMARS

In this section we describe some features of the host language attribute grammar specification to show how it is combined with the attribute grammar specification of a language extension, in this case the SQL extension, to create a specification for an extended language. Section 4.1 introduces attribute grammars and a portion of the specification of the host language. It also introduces an extension to attribute grammars, called *forwarding* that is useful in defining modular and composable language extensions. Section 4.2 shows a portion of the attribute grammar that defines the SQL language extension and discusses how it integrates SQL into Java. For presentation reasons, we have simplified and omitted some features of the specifications that do not aid in understanding how language features can be added as composable extensions.

4.1 Host language specification

Attribute grammars add a layer of semantics to the syntactic specifications provided by context free grammars by associating *attributes* with non-terminal symbols and attribute defining functions with productions. Attribute grammars as originally defined by Knuth [13] can be extended with a number of enhancements that makes them more practical to use for (modular) language specification. In our framework we incorporate higher-order attributes [21] which store syntax trees (without attribute values). These allow new language constructs (trees) to be generated and passed around the original syntax tree at compilation time. We also use a mechanism called *forwarding* [19] that makes the automatic combination of different language extensions possible.

A portion of a drastically simplified version of the Java 1.4 host-language attribute grammar specification is shown in Figure 4. It first specifies the name of this grammar module. These names, similar to Java package names, use Internet domain names to ensure uniqueness, and are used when grammars, representing languages and extensions, are composed. This process is described in Section 4.3. Next it de-

fines a collection of non-terminal symbols; the non-terminal `Stmt` represents Java statements, `Expr` represents Java expressions, and `Type` for Java types. These nonterminals are used in the concrete productions to specify the parser for Java. The nonterminal `TypeRep` is used by abstract productions to represent types.

Next are specifications for the terminal symbol for identifiers (`Id`) and its defining regular expression and for semicolons. Next several synthesized attributes (`syn attr`) and inherited attributes (`inh attr`) are defined. Attributes label the nonterminal and terminal nodes in an object programs syntax tree. Synthesized attributes store information that propagates up the tree and thus productions define synthesized attributes that label their left-hand side non-terminal. Inherited attributes store information that propagates down the tree and thus productions define inherited attributes that label their right-hand side child non-terminals. The `pp` attribute of type `String` holds the pretty-print version of constructs it decorates. This attribute decorates (written `occurs on`) nonterminals `Expr`, `Stmt`, and `Type`. The inherited environment attribute `env` is a list of pairs mapping names to their type representations; it forms a symbol table that decorates `Expr`, `Stmt`, and `Type`. The `typerep` attribute decorates expressions to indicate their type and decorates type expressions (`Type`) to indicate their representation. An `errors` attribute is a list of strings.

Concrete productions (indicated by `con prod`) such in the local variable declaration production `local_var_dcl` are used by the parser generator. Abstract productions (indicated by `abs prod`) are not. Silver allows attributes to be defined in both concrete and abstract productions. The left and right-hand side nonterminals and some right-hand side terminal symbols are named. In `local_var_dcl` the nonterminal `Stmt` is named `s` and this name is used in the block of attribute definitions that follow the production signature. Some keyword and punctuation terminals symbols, defined in the manner of terminal `SemiColon` match just one lexeme and can be referenced in the production by that lexeme, as is done in `local_var_dcl`. This production defines `pp` as expected using the pretty print of the type expression `t` and the `lexeme` of the terminal `id`. Attributes values are referenced on nodes using the dot notation. It also defines its `defs` attribute to be the mapping from the name of the identifier to the representation of the type `t`. Definitions are collected according to the scope rules of the language to populate the `env` attribute. Details of this are elided.

The `while` production defines the Java while-loop. It defines the concrete syntax by specifying that a while loop is of nonterminal type `Stmt` and that it is composed of the keyword "while", a left paren, a condition of type `Expr`, a right paren, and a loop body of type `Stmt`. The `pp` attribute is defined as expected from the string literals and value of the `pp` attribute on the child non-terminals. In the specifications, square brackets denote lists and `++` denotes list, as well as string, concatenation. The `pp` attribute will label all nodes in the concrete syntax tree, but we will omit further definitions of `pp` on productions as their behavior is what is expected. The while loop also copies its `env` attribute to its children. When this is the expected behavior we will sometimes leave these copy rules out of the given specifications.

```
grammar edu:umn:cs:melt:java14;

nonterminal Expr, Stmt, Type, TypeRep ;
terminal Id / [a-zA-Z][a-zA-Z0-9_]* / ;
terminal SemiColon ';' ;

syn attr pp :: String occurs on Expr, Stmt, Type ;
syn attr name :: String occurs on TypeRep ;

inh attr env :: [ (String, TypeRep) ]
  occurs on Stmt, Expr, Type ;
syn attr defs :: [ (String, TypeRep) ] occurs on Stmt ;
syn attr typerep :: TypeRep occurs on Expr, Type ;
syn attr errors :: [ String ] ;

syn attr hostStmt :: Stmt occurs on Stmt ;
syn attr hostExpr :: Expr occurs on Expr ;
syn attr hostType :: Type occurs on Type ;

con prod local_var_dcl  s::Stmt ::= t::Type id::Id ';'
{ s.pp = t.pp ++ " " ++ id.lexeme ++ ";"
 s.defs = [ (id.lexeme, t.typerep) ] ;
 s.hostStmt = local_var_dcl(t.hostType,id); }

con prod while
s::Stmt ::= 'while' '(' cond::Expr ')' body::Stmt
{ s.pp = "while (" ++ cond.pp ++ ") \n" ++ body.pp ;
  cond.env = s.env ;
  body.env = s.env ;
  s.errors = (if cond.typerep.name != "boolean"
    then [ "Error: condition must be boolean"] else [ ])
    ++ cond.errors ++ body.errors ;
 s.hostStmt = while(cond.hostExpr,body.hostStmt);   }

con prod idRef  e::Expr ::= id::Id
{ e.typerep = lookup (e.env, id.lexeme) ;
  e.error = e.typerep.errors
  e.hostExpr = idRef(id); }

con prod booleanType  t::Type ::= 'boolean'
{ t.pp = "boolean" ;
  t.typerep = booleanTypeRep(); }

abs prod booleanTypeRep  tr::TypeRep ::=
{ tr.name = "boolean" ;
  tr.error = [ ] ; }

abs prod notFoundTypeRep  tr::TypeRep ::= n::String
{ tr.name = "NotFound" ;
  tr.error = ["Error " ++ n ++ "not found"]; }
```

Figure 4: Simplified Java host language Silver specification.

```
con prod driver
s::Stmt ::= 'register' 'driver' d::StringLiteral
{ s.pp = "register driver " ++ d.lexeme ++ ";"
  s.errors = ... check the d is valid ... ;
 forwards to ``Class.forName ( |s.lexeme| ) '' ;  }
```

Figure 5: SQL Driver construct specification.

The identifier reference production `idRef` passes the name `id.lexeme` to the `lookup` function to extract the `TypeRep` associated with that name from its environment `e.env`. If the name is not in the environment, `lookup` returns the `TypeRep` built by the `notFoundTypeRep` production. Type checking is done by name and implemented by examining the `name` attribute on a constructs `typerep` attribute. For example, in the `while` production, the name attribute on the type node of the condition expression is checked to see that it is "boolean". Type expressions (`Type`s) are specified by concrete productions and the corresponding abstract productions construct their type representations which are used to label expressions with their type.

The attributes `hostStmt`, `hostExpr`, and `hostType` are used to generate the syntax tree in which all constructs defined in language extensions have been translated to their pure Java 1.4 representations. On each host language production, the `host` attribute for its left-hand side nonterminal is defined following the pattern shown in Figure 4. These attributes are not defined on language extension productions.

As stated, we provide only some of the simplified definitions of the Java host language that have been implemented in Silver. For example, not shown are the implementation of objects and classes which are a significant portion of the specification. But they are not critical here.

4.1.1 Forwarding and language extensions

Forwarding [19] is an extension to higher-order attribute grammars that allows new language constructs in modular language extensions to be defined in terms of existing host language constructs. But it also allows the explicit specification of semantic analyses (new ones specified by the extension or existing ones in the host language) by allowing productions to explicitly specify attribute definitions. We will use the SQL driver registration construct in line 1 of Figure 2 to illustrate this; its specification is shown Figure 5.

A valid but minimal language extension could simply rewrite this construct to the semantically equivalent statement shown in line 1 of Figure 3. While this would provide an implementation for the driver construct, it is inadequate because any errors made by the programmer would be reported as errors in the generated `Class.forName` construct that the programmer did not write. Forwarding solves this problem. It allows productions to define, in addition to attribute definitions, a semantically equivalent construct that they should, in essence, be translated to. This translation does not replace the existing construct however. This construct has the same non-terminal type as the left-hand side of the defining production. If, during compilation, a node in the syntax tree is queried for an attribute for which its production does not explicitly provide a definition then that query is forwarded to the semantically equivalent "forwards to" construct specified by the production which returns its value for that attribute. This construct also inherits, automatically, the same attribute values as the "forwarding" production.

In the case of the *driver* production in Figure 5, if a *driver Stmt* node is asked for its *errors* attributes, it returns the values computed by the *driver* production. The definition of this attribute it elided but will check if the string literal is a valid driver string — in Figure 2 it is the class name `com.mysql.jdbc.Driver`. This production also explicitly defines its `pp` attribute. Now consider the `host` attributes defined on all productions in the host language specification. These attributes contain the tree of the program in which the language extension constructs are translated to their representation in the host language. This is shown for the `while` production and others in Figure 4. If the `driver` construct is queried for the value of this attribute — as it would be during compilation — this query is passed to the forwarded-to construct and its *hostStmt* value is returned. This is how the extended language program is translated to one in the host language.

In Silver, the tree that a production forwards to is constructed using the names of the productions as tree-creation functions in which the parameters are the child trees. Throughout the paper we do not show these tree constructions but instead simply give a stylized string representing the concrete syntax of the constructed tree as this makes it easier to see what a construct forwards to. These strings use different quotes (`‘‘...’’` instead of `"..."`). Also, "holes" are specified by vertical bars (`| ... |` in which values from child trees can be included in the parameterized string. For example, in the specification of the `forwards to` construct of the `driver` construct, the lexeme of the terminal `s` is the parameter to the `Class.forName` construct. The vertical bars are in essence an unquote operator.

In the current Java implementation, we do not translate programs to Java byte code. If the *errors* attribute is empty, then the extended compiler outputs value of the *host* attribute on the root of the tree. This is pure Java code in which the language extensions have been compiled down to their representations in the host language.

4.2 Specification of the SQL extension

In Section 3 we described the SQL constructs in the SQL extension and showed what examples of the new constructs translate to. In this section we cover some of the attribute grammar specification of the SQL language extension. As in the Java specification, space limitations require us to present a simplified and reduced version of the actual specifications. We focus on how the SQL extensions work with the type system of the Java specification and how errors are collected to statically check type correctness of static queries.

The productions, nonterminals, and attributes defined in Figures 5 and 6 integrate SQL into Java. The table import production `sqlImport` and the `sqlQuery` production, for example, have Java defined non-terminals on their left-hand side, but, in most cases, SQL defined non-terminals on their right. The non-terminal *SqlQuery* (used by `sqlQuery`) and its productions are defined in Figure 7. The `sqlQuery` production reports errors from `sq::SqlQuery` and forwards to the JDBC code seen in the examples in Section 3.

Table Import Statement: In Figure 6, the `sqlImport` production adds to `defs` (and thus indirectly the the environment `env`) the table name and its type representation. This representation is just the environment (again as a list) consisting of column names and type representations specified in productions `sqlColType`, `sqlColTypesOne`, and

```
grammar edu:umn:cs:melt:java14:exts:sql ;

nonterminal SqlQuery, SqlColTypes, SqlColType ;
attribute defs, pp occurs on SqlQuery,
  SqlColTypeList, SqlColType ;

-- Table import and table type specifications --
con prod sqlImport
s::Stmt ::= 'import' 'table' t::Id
            '[' cols::SqlTypeCols ']' ';'
{ s.defs = [(t.lexeme, sqlTableTypeRep(cols.defs) ) ];
  s.pp = "import table [" ++ "] ;" ;
  forwards to skip();    }

con prod sqlColTypesOne
stfs::SqlColTypeList ::= stf::SqlColType
{ stfs.pp = stf.pp ;   stfs.defs = stf.defs ; }

con prod sqlColTypesCons
stfs::SqlColTypes ::= stf::SqlColType ','
                      stfs2::SqlColTypes
{ stfs.pp = stf.pp ++ stfs2.pp ;
  stfs.defs = stf.defs ++ stfs2.defs ; }

con prod sqlColType
stf::SqlColType ::= t::SqlType f::Id
{ stf.pp = t.pp ++ " " ++ f.lexeme ;
  stf.defs = [ ( f.lexeme, t.typerep ) ; }

-- SQL Column Types --
con prod sqlIntegerType st::SqlType ::= 'INTEGER' {..}
abs prod sqlIntegerTypeRep tr::TypeRep ::=        {..}
con prod sqlVarCharType st::SqlType ::= 'VARCHAR' {..}
abs prod sqlVarCharTypeRep tr::TypeRep ::=        {..}

abs prod sqlTableTypeRep
tr::TypeRep ::= cols::[(String, TypeRep)]
{ tr.name = "SqlTable" ;    tr.errors = [ ];
  tr.tableEnv = cols ;    }

syn attr tableEnv :: [(String, TypeRep)]
  occurs on TypeRep ;

-- Sql Query integration --
con prod sqlQuery
e::Expr ::= 'using' c::Id 'query' '{' sq::SqlQuery '}'
{ e.pp = "using " ++ c.lexme ++ " query {" ++
         sq.pp ++ "}" ;
  e.typerep = ... Java class ResultSet ... ;
  e.errors = sq.errors ;
  forwards to ``|id.lexeme|.createStatement().
                executeQuery(|sq.javaExpr|)'' ;   }

-- New Java Types for Sql Exprs for dynamic queries --
con prod sqlExprType  t::Type ::= 'sqlExpr'
{ t.pp = "sqlExpr" ;
  t.typerep = sqlExprTypeRep(); }

abs prod sqlExprTypeRep  tr::TypeRep ::= 'sqlExpr'
{ tr.name = "sqlExpr" ;
  tr.errors = [ ] ; }
```

Figure 6: Productions linking Java and SQL

`sqlColTypesCons`. These productions create this environment using the `defs` attribute. This is then packaged as a `TypeRep` by the production `sqlTableTypeRep`. This environment will be used to look up the type of column names used in SQL queries. The attribute definitions of the productions for SQL types (nonterminal `SqlType`) `INTEGER` and `VARCHAR` and type representations (nonterminal `SqlType`) are elided but define their `pp`, `typerep`, `name`, and `errors` attributes in just the same manner as the Java type in Figure 4 and the Java type `slqExpr` do. The production forwards to the Java `skip` statement since it has no Java representation. An alternative implementation would be to forward to Java code that verifies, at run time, that the specified type of the table matches the schema of the table in the database.

SQL Query: Also in Figure 6, the production `sqlQuery` defines the concrete syntax for the *using ... query* construct which is a Java `Expr`. It integrates the SQL queries with Java. It defines its `pp` as expected, its `errors` are those discovered in the SQL query `sq`, and its `typerep` is the representation of the Java class `ResultSet`. It forwards to the method calls (as illustrated in Figure 3) on the connection. The parameter to the `executeQuery` method is the Java string-valued expression generated by the SQL constructs shown in Figure 7 and stored in the attribute `javaExpr`.

In Figure 7 are the non-terminals and productions that define (a small subset of) SQL expressions. Syntax errors in SQL query expressions are detected by the parser since they are written directly in the object program (between "{" and "}") and not encapsulated in strings. Type errors are computed in much the same way as in the Java host language.

The production `sqlSelect` extracts the environment `tableEnv` from the type representation of the SQL `table`. This is passed down the SQL syntax tree in the `colenv` attribute where it is used by the `sqlId` production to look up the types of column names. This type information is then used to type-check the SQL queries. In production `sqlId` the identifier `id` could be a Java identifier or a column reference. If it is found in the environment containing column names, the attribute `colenv`, then it is a column reference and its `javaExpr` is the lexeme of that identifier with wrapped in quotes as a Java string literal. For example, in line 6 of Figure 2 the id `first_name` in the `WHERE` clause would be found in the column environment `colenv`. Thus, its translation to Java is the literal `"first_name"` seen in line 6 of Figure 3. If the identifier is found in the standard environment `env` then it is a Java variable and its `javaExpr` is the Java variable with that name. For example, in line 6 of Figure 2, `s` in the `WHERE` clause would be found in the environment `env`. Its translation to Java is a variable `s` seen in line 6 of Figure 3. The types of Java variables are then converted to SQL types. The `converted` local attribute in `sqlId` presents a simplified (from the actual implementation) mechanism for doing this. Java strings and integers are converted to their SQL versions. Java `sqlExpr` types used in dynamic queries are handled differently and discussed below.

Dynamic Queries: Dynamic creation of SQL queries creates a number of challenges for statically type checking SQL queries. The basis of the problem is that the Java identifiers that contain SQL expressions (`e1` and `e2` in Figure 2) have

```
grammar edu:umn:cs:melt:java14:exts:sql ;

nonterminal SqlQuery, SqlExpr ;
attribute pp occurs on SqlQuery, SqlExpr ;

inh attr colenv :: [(String,TypeRep)]
  occurs on SqlQuery, SqlExpr ;
syn attr javaExpr::Expr occurs on SqlQuery, SqlExpr;

con prod sqlSelect
sq::SqlQuery ::= 'SELECT' fields::SqlExpr 'FROM'
            table::Expr 'WHERE' where::SqlExpr
{ sq.pp = ... ;
  sq.javaExpr = `` "SELECT " + |fields.javaExpr| +
   "FROM" + |table.pp| + "WHERE" + |where.javaExpr|'';
  sq.errors = fields.errors ++ where.errors ++
     if table.typerep.name == "SqlTable" then [ ]
     else [ "Error: table must have type SqlTable" ]
  fields.env = sq.env ;
  fields.colenv = table.typerep.tableEnv ;
  where.env = sq.env ;
  where.colenv = table.typerep.tableEnv ;    }

con prod sqlId
se::SqlExpr ::= id::Id
{ se.pp = id.lexeme ;
  se.javaExpr = if sqltype.name != "NotFound"
                then `` "|id.lexeme|" ''
                else ``  |id.lexeme|  '' ;
  se.typerep = if sqltype.name != "NotFound"
               then sqltype else converted ;
  se.errors = se.typerep.errors ;
  local javatype :: TypeRep
        = lookup(se.env,id.lexeme);
  local sqltype  :: TypeRep
        = lookup(se.colenv,id.lexeme);
  local converted :: TypeRep
        = if javatype.name == "String"
          then sqlVarCharTypeRep()
     else if javatype.name == "int"
          then sqlIntegerTypeRep()
     else if javatype.name == "sqlExpr"
          then sqlExprTypeRep()
     else notFoundTypeRep(id.lexeme);
}
```

Figure 7: SQL query and expression specifications.

```
grammar edu:umn:cs:melt:java_sql ;

import edu:umn:cs:melt:java14 ;
syntax edu:umn:cs:melt:java14 ;
import edu:umn:cs:melt:java14:exts:sql ;
syntax edu:umn:cs:melt:java14:exts:sql ;
import edu:umn:cs:melt:java14:exts:rlp;
syntax edu:umn:cs:melt:java14:exts:rlp;

import core ;
abstract production main top::Main ::= args::String
{ forwards to java_main(args, parse) ;  }
```

Figure 8: Composed language Silver specification.

the Java type specified by the production `sqlExprTypeRep`. The information about the SQL type is not present in the particular type representation scheme used here. Thus, it cannot be determined if the SQL expression is, for example, an SQL `INTEGER` or `VARCHAR`. In the current implementation of the SQL language extension we do not attempt to statically type check dynamically generated queries. Instead, Java variables of type `sqlExpr` are given, in the production `sqlId` the converted type of `sqlExprTypeRep()`. In type checking, this type deemed to be compatible with all other types so that no errors are generated for such identifiers. Although checking such queries is possible and is done by other tools, such as the JDBC Checker [9], our goals here are not to create the most sophisticated embedding of SQL into Java. They are to show how syntax and static analysis can be added to to host language to support the functionality provided by a library.

4.3 Composition of Java and Language Extension Specifications

As outlined in Section 2, programmers compose host language and language extensions with no implementation level knowledge of the language or the extensions but need only select the desired extensions. We are currently developing an Eclipse plug-in for the extensible compiler framework that supports this selection process. The plug-in will automatically generate from the list of selected extensions the Silver specification that composes the host language and selected extensions. In Figure 8 is the Silver specification that would be generated if the programmer selected the SQL extension described above and the computational geometry extension that implements the randomized linear perturbation (`rlp`) scheme for handling data degeneracies in geometric algorithms. What the `rlp` extension does specifically is not of interest here. This composed extended language has features to support both the domains of relational database queries and computational geometry. The `import` statements import the grammar specifications in the named Silver module. The `syntax` statements import the concrete syntax specifications from the named modules to build the parser for the extended language. The final three lines import utility types like `Main`. The `main` production is similar in intent to the C `main` function; here it delegates to the main production `java_main` in the `java14` host specification. Although this shows that correctly-specified extensions can be easily composed, it is possible to write Silver specifications that when composed with the host language do not result in well-defined attribute grammars. Section 5.2 discusses issues of syntactic and semantic composability of extensions.

5. CONCLUSION

5.1 Related Work

There are other languages such as SQLJ [7], Cω [3] and .NET languages like C# that use the LINQ [15] .NET project that provide a more complete embedding of SQL constructs than we have specified in the SQL language extension above. Cω and LINQ also address the mismatch between the types in SQL and the host language. For example, SQL `INTEGER` types can have the value `NULL`, but Java `int` types cannot. Other mismatches between the object view and the relational table view of data are also addressed in these languages but not in the extension described in this paper.

These are new, well-crafted, useful languages, but they are *monolithic* solutions. They cannot be extended to provide the same sort of language and analysis support to other domains. The extensible language framework presented here illustrates how such language and analysis support can be provided in an extensible manner.

Similar work has been done in extending Java with XML language constructs. For example, JWig [6] is a Java based framework that allows programmers to write XML directly in Java. The framework analysis ensures that all statically and dynamically generated XML segments are syntactically correct XML. JWig does this by using a Java parser extended with new rules for, among other constructs, XML. This parser outputs pure Java which a standard Java compiler converts to byte-code, much like our framework. JWig then analyzes the byte-code using the Java Syntax Analyser [5]. It statically verifies that the XML segments generated and used in the original JWig program file are valid XML. One problem with this approach is that errors are reported by the Java compiler on the Java code generated by JWig. This can be confusing to programmers and is something that can be avoided in our framework since extensions can define their own error-checking analysis. This problem is shared by macro-based approaches to language extension. Some modern macro systems, such as Maya [1] and JTS [2], do however provide specific error checking facilities.

There is a significant amount of work in the language processing tools community for building extensions to languages. For example, the Polyglot extensible Java compiler [16] allows Java to be extended with powerful abstractions such as pattern matching [14]. However, this system requires one to write Java code to incorporate new extensions into Java. In the extensible language framework we propose, extensions selected by the programmer are naturally and automatically composed to form a new extended compiler. MetaBorg[4] is another system that allows one to extend a host language by adding concrete syntax for objects. This system is based on StrategoXT [20] and uses strategies and term-rewriting to process programs. Specifying semantic analyses, like the error checking, is less straight forward than it is using attributes and it is not clear that different extensions can be combined automatically.

The attribute grammar community has also addressed issues of modular language design. Of particular interest are the rewritable reference attribute grammars [8] in the JastAddII system. An extensible Java 1.4 compiler has also be specified in this system. Language extension constructs are translated to host language constructs by destructive rewrites on the syntax tree. Thus *all* analysis on an extension construct must be completed before any analysis on its translation to the host language. Although forwarding is similar to rewriting, it is non-destructive so that the original tree and the forwards-to tree exist simultaneously. This turns out to be important when multiple extensions introduce new semantic analyses because a construct will need both trees — the original for its analysis and the forwards-to tree for analyses from other extensions.

The Broadway compiler [12] allows library writers to specify, for the host language compiler, how uses of library abstractions can be optimized. This tool is based on abstract interpretation. However, it does not provide means for specifying new language constructs .

5.2 Future Work

Program comprehension: While new domain-specific syntax can be useful, if each member of a development group imported their own favorite set of extensions for use in the code for which they are responsible, group members may not understand each other's code. This problem is not unique to extensible languages, since libraries can be misused in a similar fashion, although at least the syntax if not the intent of library uses is clear. A solution is to use some discipline and restrict the set of libraries or language extensions to be used on a project. For extensible languages to have real-world impact, deployment issues such as these must be addressed.

Syntactic composability: We are also investigating techniques to ensure that the combination of syntactic specifications from several extensions will work together. Grammars used by Yacc-like tools are notoriously brittle and adding productions can easily move the grammar out of the class of grammars handled by the tools. One approach being investigated is the use of GLR parsers which can parse all context free grammars. Another approach would be the use of parsing expressions grammars, which are closed under composition [11]. Other systems, such as the Intentional Programming [18] system developed at Microsoft Research, avoid this problem by building a structure editor in which programmers manipulate the AST of the program directly.

Semantic composability: One requirement to ensure that the attribute grammar of the extended language is well-defined is that extensions need to honor the attribute dependencies of the non-terminals they extend. For example, the production `sqlQuery` in Figure 6 has a Java nonterminal (`Expr`) on its left-hand side and thus can be used anywhere that an expressions would be. Thus, it cannot define attributes such that a host language synthesized attribute depends on an inherited attribute on which it did not depend in the host language specification. Otherwise some host language production with an expression as a child may not define the inherited attribute required for computing the synthesized attribute. Although the standard definedness test [19] can perform this analysis on the extended language grammar a better approach would involve a modular analyses that can be performed by the feature designer that would ensure compatibility with other extensions that also pass the modular analysis.

Java Language Specification: As mentioned earlier, the specification of the static analysis in the Java 1.4 attribute grammar must be completed so that type errors are reported on the extended language program and not on the generated pure Java code. We intend to extend this specification to Java 1.5 as many features like the for-each loop and autoboxing and unboxing can be specified as modular extensions to the Java 1.4 specification. Silver is currently available on the web at `http://www.melt.cs.umn.edu`.

5.3 Conclusions

We have shown how new syntax and new static analysis that supports the abstractions provided in a library can be

specified so that they can be easily incorporated into an extensible language at the direction of the programmer. The new syntax and analysis address drawbacks of library-based approaches to specification of new abstractions by providing a more natural syntactic representation of the abstractions and, more importantly, providing analysis to statically check for their correct use. A key characteristic of the approach presented here is that multiple language extensions can be composed, and used by the programmer, in a manner similar to how multiple libraries can be used in the same program. This differs from the monolithic approach to language extension taken in SQLJ, Cω, and in the LINQ project.

Libraries have proved to be a very successful means for specifying, packaging, and distributing new abstractions that support the needs of different user (programmer) communities. Interested parties in small domains can design, implement, and distribute abstractions that support their work. The compositional and user-driven nature of libraries has been a key to their success, and thus we support both of these aspects in the framework for language extension presented in this paper.

Acknowledgements:

We thank Phil Russel for this assistance of the specification of the SQL extension. We thank Lijesh Krishnan for his help in the development of the Java 1.4 Silver attribute grammar.

6. REFERENCES

[1] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proc. of ACM PLDI Conf.*, pages 270–281. ACM, 2002.

[2] D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53. IEEE, 2–5 1998.

[3] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in cω. In *ECOOP 2005, Proceedings of 19th European Conference on Object-Oriented Programming*, pages 287–311, 2005.

[4] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. ACM Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 365–383, New York, NY, USA, 2004. ACM Press.

[5] A. S. Christensen, A. M. ller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. Static Analysis Symposium*, 2003.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending java for high-level web service construction. *ACM Trans. Prog. Lang. Syst.*, 25(6):814–875, 2003.

[7] A. Eisenberg and J. Melton. SQLJ part 0, now known as SQL/OLB (object-language bindings). *SIGMOD Rec.*, 27(4):94–100, 1998.

[8] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proc. of ECOOP '04 Conf.*, pages 144–169, 2004.

[9] C. Gould, Z. Su, and P. Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 697–698, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. Gray. The next database revolution. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 1–4, New York, NY, USA, 2004. ACM Press.

[11] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[12] S. Guyer and C. Lin. Broadway: A software architecture for scientific computing. In R. F. Boisvert and P. T. P. Tang, editors, *The Architecture of Scientific Software*, pages 175–192. Kluwer Academic Press, 2000.

[13] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.

[14] J. Liu and A. C. Myers. Jmatch: Iterable abstract pattern matching for java. In *Proc. International Symposium on Practical Aspects of Declarative Languages*, January 2003.

[15] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM Press.

[16] N. Nystrom, M. R. Clarkson, and A. C. Myer. Polyglot: An extensible compiler framework for java. In *Proc. 12th International Conf. on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152. Springer-Verlag, 2003.

[17] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proc. of ACM POPL Conf.*, pages 146–159, 1997.

[18] C. Simonyi. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[19] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[20] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[21] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *ACM PLDI Conf.*, pages 131–145, 1990.