A relational approach to static semantic checking

A. P. van der Meer

Master Thesis
October 21, 2008

Supervisors

Drs. ing. B.J. Arnoldus
Prof. dr. M.G.J. van den Brand
Drs. H.P.J. van Geldrop - van Eijk
Dr. R. Kuiper

Department of Mathematics and Computer Science

Abstract

Correct static semantics are a crucial part of all software. This thesis investigates a relational approach to static semantic checking. This approach means that we first extract information, then analyse it afterwards. So-called facts extracted from the program under scrutiny are inserted into a database. Constraints and queries can then be used to identify errors. In this thesis, a fact extractor is presented that can extract the required information. The fact extractor is based on grammar annotations. We have chosen to focus on the Java programming language and have created a prototype that checks for a number of Java errors. We describe the database schema used in the prototype and major implementation issues we encountered. Finally, we discuss a possible application of the method in the area of templates.

Acknowledgements

I am grateful for for all the support I have received from many people in the writing of this thesis. I want to to thank prof. dr. M.G.J. van den Brand for suggesting the topic of this thesis to me and allowing me to research it as my graduation project. I thank drs. ing. B. J. Arnoldus for his advice and support during the whole project, and drs. H.P.J. van Geldrop-van Eijk for her insightful comments and criticism. Finally, I want to thank my parents and family for their support, especially during the more stressful times.

Contents

1 Introduction **5**
1.1 Problem statement . . . 6
1.2 Outline of this thesis . . . 6

2 Fact extraction **7**
2.1 Introduction . . . 7
2.2 Methods of fact extraction . . . 7
2.2.1 Regular expressions . . . 7
2.2.2 Parse Trees . . . 9
2.2.3 Parser Generators . . . 10
2.3 Fact extraction by universal parse tree traversal . . . 11
2.3.1 Introduction . . . 11
2.3.2 Tree traversal . . . 11
2.3.3 Grammar of annotations . . . 12
2.4 Conclusion . . . 13

3 Static semantics of Java **14**
3.1 Selection of checks . . . 15

4 Database schema **17**
4.1 Introduction . . . 17
4.2 Base structure . . . 17
4.2.1 Scope Table . . . 17
4.2.2 Type & Interface Tables . . . 20
4.2.3 Expression Table . . . 21
4.2.4 FieldDec, Typeinfo, Variable & Arraydec Tables . . . 22
4.2.5 Method & Parameter Tables . . . 23
4.2.6 Import Tables . . . 23
4.2.7 Modifiers Table . . . 24
4.2.8 Package Table . . . 24
4.3 Relationships between tables . . . 24
4.4 Relation between errors and tables . . . 24

5 Implementation **27**
5.1 Introduction . . . 27
5.2 SQL . . . 27
5.3 Enriching the database . . . 28

5.4 Reference errors . . . 29
5.5 Type errors . . . 31
5.6 Declaration errors . . . 32
5.7 Contradictions . . . 33
5.8 Inheritance and interface errors . . . 33
5.9 Miscellaneous . . . 33

6 Prototype results **35**
6.1 Literals . . . 35
6.2 Inheritance . . . 35
6.3 Types . . . 35
6.4 Modifiers . . . 36
6.5 Enum . . . 36
6.6 Interfaces . . . 36
6.7 Generics . . . 36
6.8 Exceptions . . . 36
6.9 Constructors . . . 37
6.10 Initialization . . . 37
6.11 Instances & Inner classes . . . 37
6.12 Control flow . . . 37
6.13 Ambiguity . . . 37
6.14 Accessibility . . . 37
6.15 Annotations . . . 38

7 Future work: Templates **39**
7.1 Introduction . . . 39
7.2 Repleo . . . 39
7.3 Base language errors . . . 40
7.4 Template specific errors . . . 42
7.5 Ambiguities . . . 43

8 Related work **44**

9 Conclusion **47**

A SDF Grammar of annotations **52**

B Java compile-time error tables **54**
B.1 Literals . . . 54
B.2 Inheritance . . . 54
B.3 Types . . . 55
B.4 Modifiers . . . 56
B.5 Enum . . . 57
B.6 Interfaces . . . 57
B.7 Generics . . . 58
B.8 Exception . . . 59
B.9 Constructors . . . 59
B.10 Initialization . . . 60

B.11 Instances & Inner classes 60
B.12 Control flow 60
B.13 Ambiguity 61
B.14 Accessibility 61
B.15 Annotations 62

C Template Listings **63**

Chapter 1

Introduction

This thesis discusses a relational approach to implement a static semantic checker. This idea was inspired by the desire to check templates for semantic errors, like incorrect variable references and method calls. A template is, in general, a program fragment that is used to create many copies of some design easily. In particular, the templates used here consist of source code with holes that can be filled in various ways. This makes it possible to create many variations, simply by changing the way the holes are filled.

Because a template is ideally used many times, it is important there are no errors in it. The advantages of a template are greatly reduced if every variation has to be debugged separately. If an error is detected in the output of a template engine, it can quite easily be unclear what the cause of the error is. It can be the template, the input data or even the template engine. The possible errors in a template can be classified as syntactical or semantical. A syntactical error means that the source code does not conform to the context-free grammar of the programming language. Repleo[5] is an example of a template engine that can check templates for syntactical errors. For this reason, we use Repleo as our example template system in this thesis. A semantic error means that the code does not have a sensible meaning that can be executed by a computer. Existing semantic checkers, in most cases part of a compiler, cannot deal with the placeholders in a template. Furthermore, they are usually aimed at one specific language, while template engines are available for many languages. Finally, there are a number of template specific semantic errors that a semantic checker not designed with templates in mind does not cover.

Semantic checking as mentioned above is quite a large area. In this thesis, we will focus on the static semantics of the Java programming language[12]. The part of the semantics of a programming language that can be verified without executing the code for the given program. Most semantic checkers restrict themselves to static semantic checks. Java was chosen as the target programming language because it is widely used and offers extensive reflection facilities. In addition, there are many template systems that focus on Java. In particular, Repleo can check the syntax of Java templates. There are certainly easier, less complex languages that could have been chosen, but that would have carried the risk that significant issues for real-world problems would be missed. If the approach can handle the complexity of Java, it should be able to handle many other programming languages. The main disadvantage of Java is that, due to the complexity of the language, covering all possible errors is not a reachable goal in the available time. This is compounded by the fact that no clear description exists what should be considered erroneous in a Java program. The Java Language Specification contains

an informal description of the language, but errors are described throughout the document, and not listed separately. Therefore, this thesis only discusses some of the more common error types, that were considered interesting cases. In particular, language features like generics and nested classes are mostly ignored in order to simplify the discussion.

1.1 Problem statement

The goal of this work is to research whether a relational approach to semantic checking offers an advantage over more traditional methods, like attribute grammars. We expect the relational approach to have an advantage in modularity and simplicity of the checks. This includes the construction of a prototype. In any program that checks for errors, soundness and completeness are important concerns. As mentioned above, completeness as in finding all possible errors in a Java program, is not possible given the time constraints. It should be possible to demonstrate that checks for individual errors are sound and complete. A formal proof is difficult to give, since the errors themselves are not formally defined in the language specification. The main questions of this thesis are: does a relational approach to static semantic checking work, can it be made sound and complete and does it offer an advantage in modularity and simplicity over attribute grammars?

1.2 Outline of this thesis

The structure of the first part of this thesis follows roughly the structure of the process of the relational approach used. The first step is fact extraction, during which data is gathered. The next step is insertion into the database. During this step some initial checks are done to ensure relational integrity. The third step is where more detailed checks take place, to test for more complicated errors.

The first step, fact extraction, is discussed in Chapter 2. Chapter 3 describes the checks that we want to implement in more detail. The fourth chapter contains information on the database schema used to store the facts once they are extracted, while Chapter 5 discusses the implementation of the checks themselves. The prototype is discussed in Chapter 6. Chapter 7 contains a discussion of potential future work, specifically relating to templates. Finally, Chapter 8 treats related work and 9 some conclusions concerning the relational approach.

Chapter 2

Fact extraction

2.1 Introduction

The source code of a computer program, or indeed almost any text contains a large amount of information. In this case, we are interested in some specific parts of that information, but not all of it. The solution is fact extraction. In the fact extraction process, small packages of information called facts are constructed based on the contents of the text. In principle, it can be done with any text, but here, we will focus on extracting facts from Java source code. For example, consider the example program in Listing 2.1.
In Listing 2.1, source code is listed, with comments indicating some examples of places where facts can be extracted. The term "fact" is used here to describe several pieces of information taken from the source code and combined into a single item. For example, the "variable declaration" fact might consist of the name of the variable, the type, the modifiers and where it is declared. It must be noted that a fact is rarely useful in isolation, so nearly all facts will reference other facts to indicate the relations between them.

In order to extract the desired facts from source code, we need to identify which information each fact should contain and where that information can be found. Ideally, these two concerns should be orthogonal, so that the choice of the extraction method has no influence on the definition of facts. In practice, each extraction method imposes its own set of limitations on the way facts are defined. For example, in some fact descriptions consist of code that constructs the desired facts, while others are more declarative. Therefore, this chapter starts with a discussion of various fact extraction techniques, followed by a more extensive description of the chosen method and how facts are defined in it.

2.2 Methods of fact extraction

2.2.1 Regular expressions

A simple method of extracting facts from given source code is looking for specific patterns, described by regular expressions. Regular expressions are used by regular expression processors to examine texts and find matches. For example, if an assignment consist of a variable name followed by an assignment symbol and then an expression, it is possible to construct a regular expression to find that pattern without constructing the whole parse tree. However, the source code may contain occurrences of the pattern, for example in string constants, that

Listing 2.1: Fact extraction example

```
1  Package Customer; // Package declaration
2  import java.util.*; // Import declaration
3
4  public class CustomerData {// Class declaration
5      private Random randGen = new Random(); // Class reference,
6                                //constructor call, variable declaration
7      private String name; // Class reference, variable declaration
8      int id; // variable declaration
9
10     int SetName(String customerName) { // Method Declaration
11         id = randGen.nextInt(); // Variable assignment,
12                                 // variable reference, method call
13         name = customerName; // Variable assignment, variable reference
14         return id; // Variable reference
15     }
16
17     String GetName(int customerID) { // Method declaration
18         if (id==customerID) { // Variable reference
19             return name; // Variable reference
20         }
21         else {
22             return null; // Constant expression
23         }
24     }
25 }
```

are not actually assignments. This would lead to a false positive. An example of a fact extractor that uses this technique for function calls is mkfunctmap[15]. Mkfunctmap is a Perl script that scans C code for function calls. The advantages here are that the resulting fact extractor is quite small and easy to implement, as long as only a limited set of simple fact types has to be extracted. The approach also has several drawbacks. The fact extractor is not very flexible and without the full information of the parse tree, the chance of false positives and false negatives is quite high[25].

2.2.2 Parse Trees

Based on the previous section, we conclude that pattern matching based fact extraction is not suitable for our needs. We want to extract a variety of facts, which would likely require a separate pattern each. In principle, this is not unacceptable, but in order to avoid significant false positives and false negatives, we would need patterns that are quite large and complex. Simple patterns are just not powerful enough to express what is required to avoid errors. A solution is to move from patterns to grammars. Grammars are a more more powerfull and precise method of describing languages. A grammar can be used by a parser to parse a text, creating a so-called parse tree that describes the structure of a text in terms of the grammar.

Compiler parse trees

One tool that usually constructs a parse tree is a compiler. An advantage is that precisely identifies the language that the fact extractor accepts, namely to exactly the language that the compiler accepts. For a language like C, with many dialects, that can be very useful.

On the other hand, it is not always possible to access the parse tree created by the compiler. While for example the gcc compiler from version 3 onwards can dump parse trees from several phases of the compilation process, earlier versions require modification[3]. Many commercial licenses prohibit program modifications like this. Furthermore the parse tree format itself might well be protected by copyrights and patents. Another problem is that what is useful for a compiler is not always useful for fact extraction. The compiler needs details of system libraries to create object code, like for example memory sizes of types, while for fact extraction those are usually irrelevant. Finally, the reliance on a specific compiler reduces the flexibility of the fact extractor.

Custom parsers

As mentioned earlier, a full parser is necessary to extract all information needed to check errors. Since a parser written to be used in a compiler does not really output the data in a useful form, an obvious solution is to create a separate parser. This is a common approach in the reverse engineering, used in tools like Rigi[31] and CIA++[13] for C/C++. Most of these tools are language-specific, though there are examples, like Columbus[11], that are intended to be more universal. In the case of Columbus, this is achieved using plugins. It should be noted that the set of C plugins, called CAN, appears to be the most-used and most-developed.

The main disadvantage of tools like these is that they are mostly intended for reverse engineering, and do not provide output that can be used directly for our purpose. The advantage is that the parser for the target language is already made and tested, but we would need a way to process the output of this parser. Since most custom parsers support only one input language and have their own output format, that does not seem a big improvement.

2.2.3 Parser Generators

If it is not worth the effort to adapt or adapt to a custom parser, we could try generating a custom parser instead. That is the province of parser generators like Yacc[17], Antlr[26] or JavaCC[20]. Closely related to this are universal parsers. In fact, what is generated by most parser generators is a parse table that is used by a corresponding universal parser to parse the desired grammar. The input of parser generators consists of a description of the grammar with production rules, combined with so-called semantic actions that are executed by the parser whenever it finds a node of the appropriate type. We could use this facility to construct the appropriate fact or facts for each node. The advantage is that this approach is quite powerful, since we construct facts using a full imperative language. This comes at a price, the complexity is quite large. Creating references to other facts, for example, does not seem possible without adding our own administration. While we can of course use subroutines and libraries to streamline the definitions, but it still seems more complicated than necessary.

JastAdd

Another approach to parser generation is JastAdd[10]. JastAdd is described as a system for the modular description of compiler tools and languages. It is based on a combination of Java with abstract syntax trees (AST). Each nonterminal is represented by an abstract class. In turn, each production rule that corresponds to a given nonterminal is represented by an actual class, that extends the nonterminal class. Finally, in the AST each node is represented by an object, of the class that corresponds to the production rule that was used to parse it. The grammar can be extended by adding extra (abstract) classes to it, and behaviour can be added to existing classes by defining aspects. In addition, JastAdd has some declarative features in the form of attributes and rewriting. While the latter is of no interest to the present discussion, the former provides the support for references that YACC and JavaCC lacked. To be precise, attributes in JastAdd are based on ReCRAGs, Rewritable Circular Reference Attribute Grammars. As the name suggest, these allow for the references we need. The disadvantage is that while it is possible to define new grammars in JastAdd, this is an extensive job and most existing grammars seem to focus on Java. Also, while aspects allow for code to be reused, it is likely a separate aspect would have to be written for each fact, which is not very convenient.

ASF+SDF

Another parser generator that deserves mentioning is ASF+SDF[8]. ASF+SDF is presented as a language definition formalism. It consists of a Syntax Definition Formalism (SDF) to which a term rewriting language, Algebraic Specification Formalism (ASF) was added to express semantics. We will discuss each of these components below.

SDF is a formalism intended for the definition of grammars for a variety of computer-based languages, even ones that were not designed with grammars in mind. A grammar written in SDF describes the syntax of a language with production rules, possible with annotations. Annotations can be used to add information to the parse tree. For example, it is common practice to add a so-called cons-annotation to each production rule, to simplify their identification. The grammar rules can be split into several modules, that are combined in the final grammar. This way, parts of grammars can also be reused. Once a SDF grammar has been

created for a language, this definition can be used to generate a parse table. The scannerless generalized LR parser SGLR can parse terms using any parse table created in this way. The output of this process is the corresponding parse tree of the term in UPTR[33] format, complete with annotations from the grammar added to the appropriate nodes.

In order to add semantics to and manipulate the parse trees created by the SGLR parser, ASF was created. ASF takes a parse tree as input, applies a set of equations and produces an output parse tree. This process can be used to define a fact extractor: the equations that form a traversal rewrite the parse tree to the desired facts. This approach is used to collect the facts for RScript[18]. It can be used for any SDF grammar and the modularity makes it is possible to have a set of equations for each fact and merge the results later. The drawback is that the equations can get quite complicated. This can (in part) be solved by increased modularity, but that carries a significant penalty in performance, due to the overhead of extracting each fact separately. If, for example, twenty types of facts each have their own traversal, the entire code has to be traversed twenty times. Furthermore, it is not easy to create references between facts without losing a lot of the modularity.

Based on this, we can conclude that ASF is not really suited to our needs. That does not mean that we cannot use SDF separately, and indeed, SDF offers several attractive properties. There are SDF grammars of a number of languages, the same grammars are used by the Repleo template engine[5] that formed part of the inspiration for this project. The annotations provide a way to define facts in a direct, declarative way. We do need a separate implementation of a tree traversal to find the annotations in the tree and determine the appropriate values. This will be discussed in the next section, together with the annotations that describe the facts.

2.3 Fact extraction by universal parse tree traversal

2.3.1 Introduction

We now have determined that we want to use SDF grammars with annotations to define facts. There can be other annotations added to the grammar, like constructors, but we will ignore those for moment. This section first describes the (simple) tree traversal that is used to locate annotations in the tree, and then the grammar of the language that is used to write annotations.

2.3.2 Tree traversal

The annotations inserted into the parse tree describe how a particular type of node must be treated. In order to extract all facts from a parse tree, the fact extractor must check every node in the tree for annotations. This is done with a tree traversal. The traversal starts in the top node of the tree. The extractor first checks if the node has any *extractfact* annotations. If that is the case, the required facts are created and filled with the requested information from this node. The fact is then emitted by the extractor. The traversal proceeds to visit each of the child nodes, and so on. If any new information has to be added to facts that already has been emitted, an update is issued. This means the receiver of the facts must be able to locate the fact to be updated. The current implementation assumes that a field called *id* can serve as a key, because we insert the facts directly into a database. Other solutions are certainly possible.

2.3.3 Grammar of annotations

The fact extractor constructs a list of facts based on annotations in the grammar. An example of such an annotation is:

```
VarDecId -> VarDec {cons("VarDec"),
                    extractfact("variable","new",
                      ["$self.getID->ID","fielddec.ID->declaration",
                       "$0.toString->name", "$self.getOffset->position",
                       "scope.ID->scope"])}
```

This annotation could be used for the variable declaration fact from Listing 2.1. Upon reaching the VardDecId node with this annotation, the traversal creates a new record of type variable. It then creates five fields, ID, declaration, name, position and scope, and inserts the appropriate values into them. Some, like `fielddec.ID` in the example, are based on values extracted earlier, others, like `$self.getID` or `$0.toString`, are filled with data from the parser or the extractor. The annotations that trigger the traversal have the following form:

```
extractfact(type,action,fields)
```

type describes the type of the fact.

action describes whether a new fact is created or the fields are added to an existing fact. In the latter case, the fact extractor decides which fact the update is applied to based on the type of the field and a field called *id*, assuming it exists. New facts are created before the fact extractor visits the subtrees, updates are handled after the subtrees have been processed. We will discuss the reasons later.

fields contains a list of field descriptions. Field descriptions describe the actual content of a fact. A field description consist of a source and a target. During the traversal, the source is evaluated and added to the fact with target as a label. A source can be a constant, which can be put into the fact directly. Another option is to refer to a field of a fact that has been created earlier. This is done by naming the type of fact and the field name. A final option is to use a node and an operation. These can be used, for example, to get the value of a node, to get a position that is related to a fact or to relate different facts:

- **toString** returns the string that is represented by the node.
- **getID** returns an unique identifier assigned by the parser.
- **getLevel** returns the number of facts of the current type between the node and the root. It can be used to determine the depth of the node in the tree, and is used for example in Section 4.2.3 to fill the expression table.
- **getScope** returns a number that can be used for scope calculations. Details of this are discussed in Section 4.2.1. A major property of this operation is that the value returned depends on when it is executed during the traversal. In order to get both numbers necessary for comparisons, one has to be retrieved in a new fact and one has to be added by an update. This is the reason updates are done ofter the children of a node have been visited: to create an opportunity to get the necessary data.

getCons returns the value of the cons annotations, that is added to the production rules of many grammars when the grammar is created. These are often added to a grammar by its authors. This function exists mostly for completeness reasons and it is not used in the prototype implementation.

getPosition and derived operations return position information about the node in the original file. GetPosition returns a string that describes all details, individual components can be extracted with the other operations.

A SDF grammar of the annotations can be found in Appendix A.

2.4 Conclusion

In this chapter, we have discussed the fact extraction process that provides the fundamental data for the static semantic checks. We have concluded that a fact extractor based on SDF grammars using annotations gives us the flexibility and ease of use. Fact definitions can be quite compact, because they are declarative in nature. We have implemented fact extractor in Java that traverses the parse tree and processes each annotation.

Chapter 3

Static semantics of Java

Before we can check the static semantics of any language, we have to define what static semantics actually is. The semantics of a programming language is that part of the language that is concerned with the meaning of programs when executed. Static semantics refer to the part of semantics that can be analysed without actually executing the program. A static semantics analyser checks whether the meaning of a program can be properly determined. For example, a wrong variable reference means a necessary value cannot be determined when needed.

The static semantic checker concept developed in this thesis originally was intended for software templates. The set of errors in a software template is a combination of the errors in the static part of the template combined with the errors in the template code. Due to time constraints, it was not possible to look at all possible programming languages and their errors. As programming language, we choose to restrict ourselves to Java. Java offers an extensive reflection system that should make gathering context information easier. In particular information about code that is not directly part of the program itself, like libraries that are accessed, can be acquired easily. Furthermore, Java is statically typed, which allows the investigation of typechecking.

The second restriction we chose was to check pure Java programs, without template code, first. If the checker cannot deal with that, it obviously cannot deal with the added complexity of the template code. Chapter 7 of this thesis discusses template-related issues in more detail. Even with this restriction, Java is still a very large programming language, with a number of complex features.

In order to simplify the development and presentation of the errors, we choose to ignore nested classes, annotations and generics. The first feature is not widely used, and it complicates the treatment of classes and references. Additionally, the checks that specifically deal with nested classes are rather complicated and obscure. Annotations are rather new and specifically defined not to affect the execution in any way. Finally, generics are more widely used, but again they complicate the checks and part of their intended purpose is covered by templates.

After all these restrictions, we can discuss what will be treated in this thesis. One problem is that there is no clear, self-contained list of Java errors. The Java language definition[12] lists a large number of compiler errors, but they are scattered all over the text and do not have a formal definition. While there is a more formal definition of the semantics of Java[2], it does not include a definition of errors. There is a set of Java Compatibility Test Tools[27],

that include compiler compatibility and presumably tests for compile-time errors. The tools have been developed by the Java Community Process, and unfortunately access is limited. Because of this, we have chosen not to use the Test Tools.

In order to make sure that the selection of errors discussed in this thesis is a representative sample, we have to classify the errors. Examples of all classes of errors can be found in listing 3.1. We have also created a list of Java errors based on the language description, which can be found in Appendix B.

3.1 Selection of checks

Type errors The static type system of Java is one of the main features of the language. All expression must be assigned a type, otherwise successful compilation is impossible. Since expressions form the main part of a Java program, any serious static semantic checker has to check for this type of errors. One specific issue with the type system is that, as the name implies, it is a system, where each part depends on the other parts. An example of a type error is in the if-statement in listing 3.1 in line 16. There, the expression in the guard of the if-statement is an assignment. In Java, assignment are considered expressions that have a type as any other, so this is syntactically correct. The type of an assignment is defined as the type of the variable that is assigned to, in this case an *int*, while the guard can only be a boolean expression.

Reference errors The actual code of a Java program contains a multitude of references, to variables, to methods and to types. If the target of the reference cannot be found, or there are multiple equally valid targets, the compiler needs values that it cannot find. This is an area where a relational approach offers an advantage, since each reference is in effect a relation between two pieces of code, represented by facts. The example program in listing 3.1 contains two reference errors, one in line 5 that references a constructor that does not exist, and one in line 10 that references a variable that does not exist.

Declaration errors Just about every Java program declares classes, interfaces, methods and/or fields. Every declaration needs a unique identifier in order to reference without ambiguity. An obvious example is two variables in the same scope that have the same name, like in lines 14 and 15 of the example in listing 3.1. It is not as simple as that, since Java allows for a certain amount of hiding and overriding, but errors like these can still be easily made.

Contradictions Each Java program consists of at least one class, with one or more methods. Each of these has a number of properties, like visibility, that are determined by its declaration. This class of errors deals with problems that arise when the declaration provides multiple values for a certain property, or does not meet the requirements for one it claims to have. An example of the first is a method that is declared to be `public` and `private`. In that case, it is not clear who is allowed access to the method. One example of the second is a method that is declared to return an integer, but can execute without returning a value at all. The method $SetName$ in listing 3.1 is an example of this. In that case, the method does not meet the requirements of a property it claims to have. Another example is a method that is declared both `abstract` and `final`. No method can meet the requirements for both these properties at the same time, so the combination makes no sense.

Inheritance and interface errors Nearly any Java class[1] has to extend exactly one other class, called the superclass, and can be implementing any number of interfaces. This is not unrestricted, for example, a class cannot extend itself. These errors are related to reference errors, because a class that is extended has to exist and allow itself to be extended. Because there are some specific restrictions, however, it makes sense to create a separate class for this group of errors. The class in the example program in listing 3.1 obviously tries to extend itself, which is an inheritance error.

Miscellaneous Finally, there are some errors that do not really fit in the other classes, but do not really merit a class of their own. Most errors in this class deal with literals and exceptions. Other examples include control flow errors, like unreachable code. By definition, these errors are only loosely related. The initial value that is assigned to the variable id in the example program is too big to fit in a Java integer, which are capped by the language definition at 2^{31} (2147483647), so that is a miscellaneous error.

Listing 3.1: Error example

```
1  Package Customer;
2  import java.util.*;
3
4  public class CustomerData extends CustomerData {// Inheritance error
5      private Random randGen = new RAndom(); // Method reference error
6      private String name;
7      int id = 7777777777; // Misc. error
8
9      int SetName(String customerName) { // Contradiction error
10         id = randen.nextInt(); //Variable reference error
11         name = customerName;
12     }
13
14     String GetName(int customerID) {
15         int customerID // Declaration error
16         if (id=customerID) { // Type error
17             return name;
18         }
19         else {
20             return null;
21         }
22     }
23 }
```

[1] The exception being Java.lang.object, the primordial class.

Chapter 4

Database schema

4.1 Introduction

Fact extraction is the first step in a larger process, not a goal in itself. In fact, it may be the case not all data that is needed for the analysis of the program can be extracted by fact extraction. Java is an example of this, because for example on-demand imports cannot be resolved without reflection. In a language like C, this would be less of an issue, because the preprocessor adds the text of all imports to the source code, so the fact extractor can access it. Before the fact extraction starts, we have to determine which facts can and have to be extracted. Once the fact extraction from a program is complete, the rest of the data needed is acquired and the facts are processed to identify errors. Ideally, the structure of the facts extracted is such that the desired information can be extracted easily. However, the annotations that define the facts to be extracted follow the structure of the grammar. Since in our current setup, the facts are inserted directly into the database, this influences the way the data is stored. The data cannot always be gathered in a way that is the most convenient for the implementation of checks. In this chapter, we will describe the structure of the tables in the database we use to store the information from the fact extractor. Figure 4.1 shows the overall structure of the database.

4.2 Base structure

4.2.1 Scope Table

One of the central tables in the database is the scope table. Details of the scope table fields are described in Table 4.1. Information about scopes is needed to resolve references correctly. In a syntactically correct Java program, scopes are always properly **nested**, so they form a tree structure. In order to determine if for example a reference can refer to a declaration, it must be inside the scope of that declaration. In order to simplify this process, the scope table contains all the scope information of the program. In particular, the scope field contains both numbers needed for nesting comparison, in order to make usage in constraints possible. Other tables reference the scope table to indicate their place in the tree. In addition to the nesting, some declarations are allowed to **hide** others. This means some additional information is needed, to indicate if this is allowed. We discuss the nesting problem first.

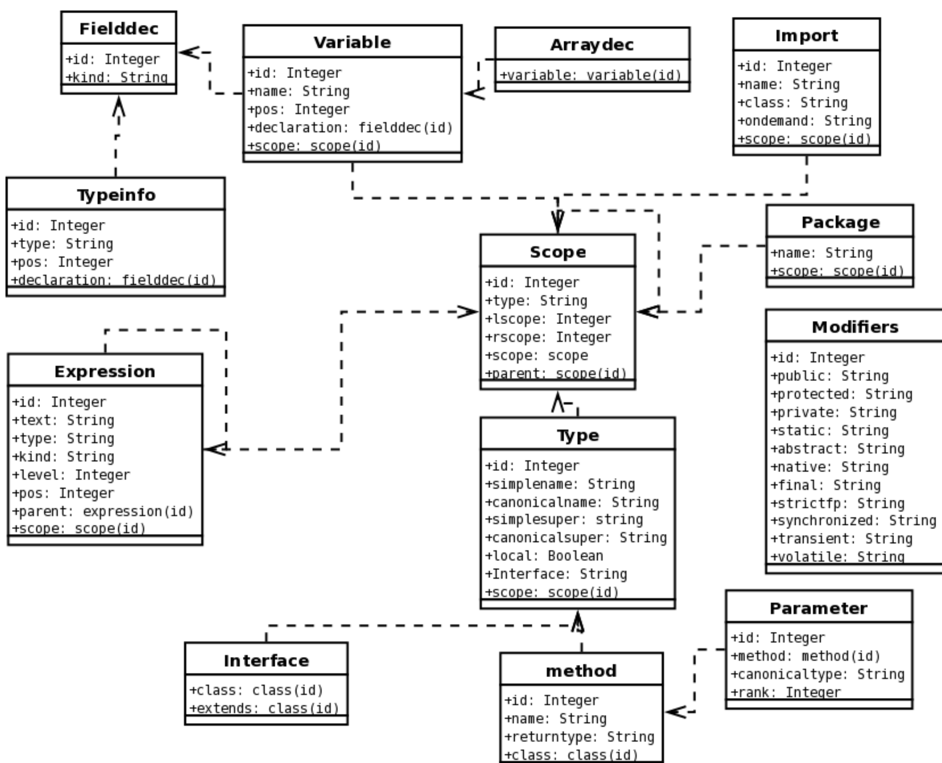

Figure 4.1: The main structure of the database.

Image	Field	Description
	id	key
	type	type of scope
	lscope	begin of scope
	rscope	end of scope
	level	level of scope
	scope	combined scope info
	parent	parent scope

Table 4.1: Scope table details

Nesting

One issue is that facts are emitted in a linear fashion. Any tree information must be added to the facts explicitly. The simplest way to extract this is to give each scope a reference to its parent scope, as in Figure 4.2. This extracts the full structure of the program, so all information is there. The disadvantage is that the question whether one scope contains another cannot be answered directly. It requires knowledge of all ancestors, so the parent relation has to be traversed recursively. This is a task that databases are not really suited for and a significant performance penalty seems inevitable, because scope comparisons are likely to occur frequently. So, it is worthwhile to add information to the scope table if that can be used to speed up that kind of operation. Another possible method is to construct paths

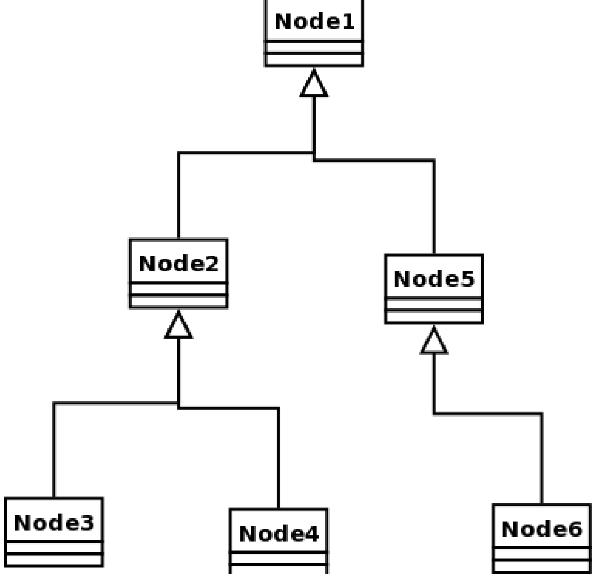

Figure 4.2: Tree with parent references

describing all ancestors for each scope, as in Figure 4.3. The paths can be constructed inside the database by repeatedly selecting the nodes that do not have paths yet but have parents that do. For each node, a path can then be constructed based on the path of the parent. If the path of a scope is a prefix for the path of another scope, then the first scope contains the second, otherwise it cannot contain it. This is similar to regular string comparison which all database systems should be able to perform. The efficiency does depend on the length of the path, since the longer the paths get, the longer it takes to compare them. This highlights another, slight, disadvantage: the size of the path, like the number of ancestors in the first solution, cannot be predicted beforehand. The larger the program, the longer the paths are likely to be, but that is only a rough guideline. This could lead to problems if the paths do not fit in the space provided. The table can be defined such that paths can be any size in the database, but at the cost of a speed penalty, because another layer of indirection is needed. The third method is to give each scope a pair of numbers that mark its begin and end, as in Figure 4.4. The numbers can be generated based on the traversal or based on position information, as long as the rule holds that a scope begins before any scope it contains and ends after any scope it contains. The question how two given scopes relate can then be answered trivially, by determining if the begin of one is smaller than the begin of the other,

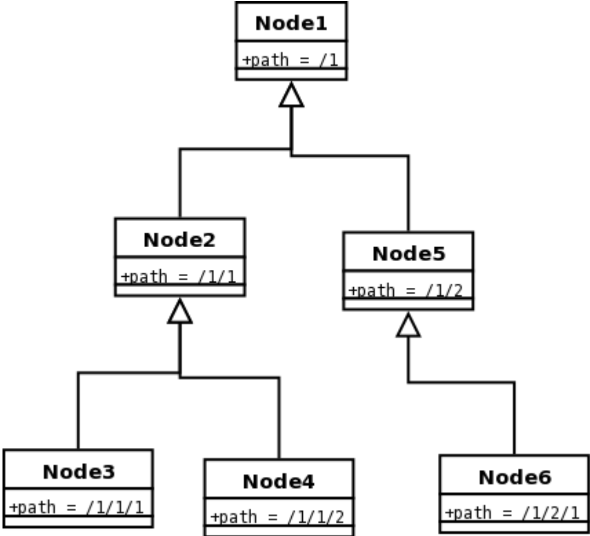

Figure 4.3: Tree with parent references and path strings

and vice versa for the end. This is easily implemented and obviously takes only constant time to calculate. Note that we assume the structure is static, since any new nodes that are added could invalidate large parts of the tree at once.

Hiding

However, in Java, some variable declarations are allowed to hide others. That means that knowing that the scope of one variable declaration is in the same scope as another one with the same name is not enough to declare it illegal. Hiding also complicates resolving variable references, because there can be multiple candidate declarations. If we traverse the references between the scopes each time, we can solve this by simple stopping at the right time. However, if we use paths or numbering, we don't have enough information to do that. A solution to this is adding a level to each scope. By giving scopes that do not conflict different levels, the database can determine if two variables conflict. In the current situation, there are only two levels of scope: class scopes and method scopes. A variable declared within the scope of a method can hide a variable that is declared in the scope of a class. So, only two levels are needed, and they can be assigned by the fact extractor directly. If we have to consider nested classes, a separate fact is necessary to determine the current level. That information can then be inserted into the scope facts.

4.2.2 Type & Interface Tables

The Type table, described in Table 4.2 stores information on classes and interfaces in the database. This may seem surprising, since classes and interfaces are similar, but not synonymous. However, in Java, classes and interfaces are treated the same in many places, including reflection. So, it is more convenient to treat them the same, and in this case, the fact extractor can accommodate that. In addition to storing both classes and interfaces, this table also stores information about local types, that are located in the file from which the facts

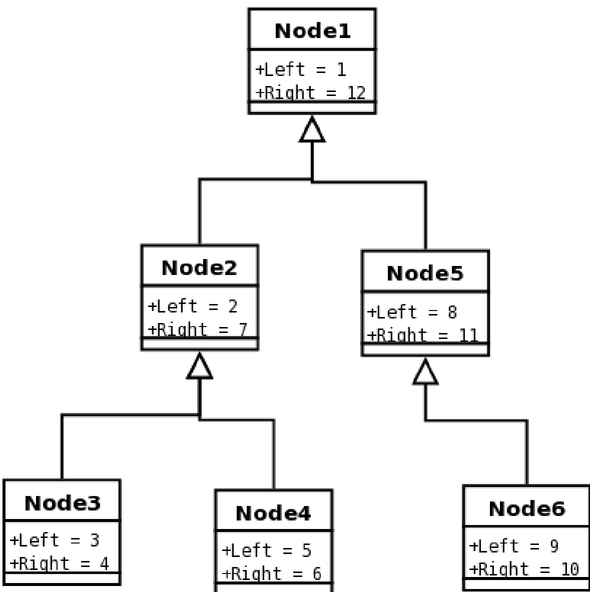

Figure 4.4: Tree with parent references and scope numbers

have been extracted, and nonlocal types, from outside the file. The nonlocal information is all gathered by reflection, not by the fact extractor, so it could easily be put in its own table. However, since it doesn't matter to Java where a given type is located, it makes more sense to keep all types in the same place. One issue is that the full canonical name of a type cannot be determined by the fact extractor. That name is necessary to refer to classes unambiguously. So, the database has to be enriched with this information, gathered by extraction, before checking can begin. As mentioned above the table Interface is not the equivalent of class for interfaces, but instead it contains information about implementation of interfaces. This creates a minor nomenclature problem: in Java, classes *implement* interfaces, while interfaces *extend* other interfaces. Since the Class table contains both interfaces and classes, neither is the obvious choice to use to indicate the interface implemented/extended. In this case, *extends* was chosen over *implements* because it felt more appropriate, but other choices are possible.

4.2.3 Expression Table

Java is, like C and C++, an expression-oriented language. Thus, the expression table forms a core part of the schema. Details are in Table 4.3. The text field of each expression contains its text, taken from the source code. This value is mostly used in case of variable references, where the name of the variable referenced is needed. Other uses include the value of literals. Whether an expression is a variable reference can be determined by its kind. The kind field describes the nature of the expression represented, for example a constant, a variable reference or an addition.

The most important property of an expression for static semantic checking is its type. Apart from a few basic cases the type of an expression cannot be extracted by the fact extractor, so computing the types of all other expressions is one of the main tasks of the semantic checker. For many expressions, the type depends on the types of its subexpressions.

Image	Field	Description
	id	key
	simplename	simple name of type
	canonicaltype	canonical name of type
	simplesuper	simple name of supertype
	canonicalsuper	canonical name of supertype
	local	true if the class comes directly from the source file
	interface	true if the type is an interface
	scope	reference to scope class sits in
	id	key
	class	identifier of type that extends
	extends	identifier of type that is extended

Table 4.2: Type and Interface table details

Image	Field	Description
	id	key
	text	text of the expression in the code
	type	canonical type of the expression
	kind	kind of expression
	level	level of expression
	pos	position of expression in the code
	parent	parent expression
	scope	reference to the scope the expression is in

Table 4.3: Expression table details

The parent links can be used to collect these types, if they have been computed. This implies that expressions must be typed in a strict order, such that all subexpressions have been typed before a parent expression is processed. For this end the level field is used. This contains information extracted about the position of the expression in the tree. By ensuring that the level of an expression is always lower than any of its subexpressions, the checker can ensure that the correct order is maintained.

4.2.4 FieldDec, Typeinfo, Variable & Arraydec Tables

These tables are discussed together because of their strong connection, with details in Table 4.4. One of the basic principles of the fact extractor is that there can be only a constant number of facts per node. However, in Java, it is possible to declare multiple variables in one, shorthand, declaration. To bridge this gap, the information is split over several facts. Variable contains information on the names of the declared variables, fielddec links them to the corresponding type and typeinfo contains the actual type information. If generics, where types can be parameterized, were to be added to the checker, this part of the schema would have to be extended. The main difficulty is that the canonical names of the types used as arguments have to be determined to complete the canonical name of the generic type itself. Typeinfo does not contain the required information at this point.

Image	Field	Description
Fielddec +id: Integer +kind: String	id kind	key kind of declaration
Typeinfo +id: Integer +type: String +pos: Integer +declaration: fielddec(id)	id type pos declaration	key type of declaration position of declaration in source code reference to the declaration corresponding to this type
Variable +id: Integer +name: String +pos: Integer +declaration: fielddec(id) +scope: scope(id)	id name pos declaration scope	key name of variable position of declaration in source code reference to the declaration corresponding to this type reference to the scope of this variable
Arraydec +variable: variable(id)	variable	reference to the base variable

Table 4.4: FieldDec, Typeinfo, Variable and Arraydec table details

Finally, Arraydec contains information about the dimensions of declared arrays. Since it is not possible to declare arrays with a set range in Java, the fact extractor can do little more that emit one fact per array symbol it finds with a link to the corresponding variable. The checker can determine the number of dimensions of each variable by counting the corresponding records in the Arraydec table.

4.2.5 Method & Parameter Tables

The Method table stores information about the methods declared in the classes and interfaces in the Type table, both local and non-local. The two main properties of a method stored here are its name and its return type. The third part, the parameters that are needed to invoke it, is stored in the Parameter table. Details of the tables can be found in Table 4.5. The Parameter table is unusual in that all information in it is gathered from either reflection or the Variable-table group, not extracted by the fact extractor. For local classes and interfaces, the parameters are first extracted in the same way as the rest of the variable declarations. The parameters can then be identified by using the FieldDec kind property, and by using the pos fields the ordering can be identified.

4.2.6 Import Tables

Types in Java are usually not referred by their full canonical name, but by their simple name. Imports are needed to determine the full name for those cases. Fortunately, the full name declared in the import can be extracted in parts, so single imports can be resolved directly by comparing the simple names to the ones given. The on-demand imports require reflection, to determine which classes and interfaces are available in the package that is imported.

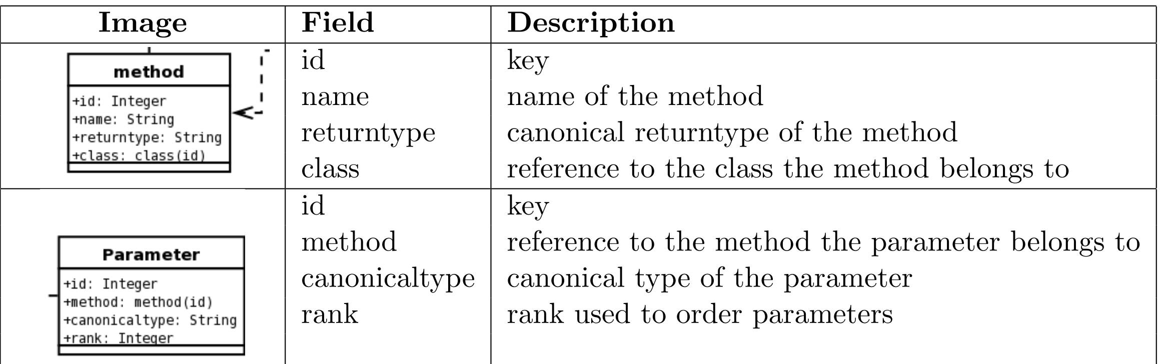

Image	Field	Description
method +id: Integer +name: String +returntype: String +class: class(id)	id name returntype class	key name of the method canonical returntype of the method reference to the class the method belongs to
Parameter +id: Integer +method: method(id) +canonicaltype: String +rank: Integer	id method canonicaltype rank	key reference to the method the parameter belongs to canonical type of the parameter rank used to order parameters

Table 4.5: Method and Parameter table details

Image	Field	Description
Import +id: Integer +name: String +class: String +ondemand: String +scope: scope(id)	id name class ondemand scope	key canonical name of imported type simple name of imported type indicates if the import is ondemand reference to the scope the import is in

Table 4.6: Import table details

4.2.7 Modifiers Table

Most declarations in Java can have several modifiers. The Modifier table has a field for each kind of modifier. Details of the table can be found in Table 4.7. Since many modifiers are applicable to several types of declarations, they are grouped together in the grammar and have to be extracted together. This means the Modifiers table, though not directly linked to one specific table, in fact references several tables: Class, Variable and Method.

4.2.8 Package Table

The Package table is the smallest of the tables, because there can be only one package declaration in a given file. That declaration is, however, necessary to determine the canonical name of the local classes and interfaces. Details of the table can be found in Table 4.8.

4.3 Relationships between tables

Table 4.9 shows the direct relations between the various tables and the reasons behind them.

4.4 Relation between errors and tables

Table 4.10 shows how error categories are related to the database tables. As can be seen in the grid, most database tables are used by most, if not all error groups. This is not surprising, since the categories were grouped on error characteristics, not on the structure of Java. There are several exceptions, like import. The import table is used in the beginning of the checking

Image	Field	Description
Modifiers +id: Integer +public: String +protected: String +private: String +static: String +abstract: String +native: String +final: String +strictfp: String +synchronized: String +transient: String +volatile: String	id	key
	public	indicates presence of public modifier
	protected	indicates presence of protected modifier
	private	indicates presence of private modifier
	static	indicates presence of static modifier
	abstract	indicates presence of abstract modifier
	native	indicates presence of native modifier
	final	indicates presence of final modifier
	strictfp	indicates presence of strictfp modifier
	synchronized	indicates presence of synchronized modifier
	transient	indicates presence of transient modifier
	volatile	indicates presence of volatile modifier

Table 4.7: Modifiers table details

Image	Field	Description
Package +name: String +scope: scope(id)	name	package name
	scope	reference to the scope of this package

Table 4.8: Package table details

Table	Linked to	Cause
Typeinfo	Fielddec	Each type has a declaration
Variable	Fielddec	Each variable has a declaration
Arraydec	Variabledec	Each array declaration corresponds to a variable
Variable	Scope	Each variable has a scope
Expression	Expression	Most expressions have a parent
Expression	Scope	Each expression is in a scope
Package	Scope	Each package has a scope
Type	Scope	Local types have a scope
Interface	Type	Some types extend interfaces
Method	Type	Each method belongs to a type
Parameter	Method	Each parameter belongs to a method

Table 4.9: Relations between tables

	Type	Reference	Declaration	Contradiction	Inheritance & Interface	Misc
Scope	X	X	X	X		X
Class	X	X	X	X	X	X
Interface	X	X		X	X	X
Expression	X		X	X		X
Fielddec	X	X		X	X	X
Typeinfo	X		X	X	X	X
Variable	X	X	X	X	X	X
Array	X	X		X	X	
Method	X	X	X	X	X	X
Parameter	X	X	X	X	X	
Import	X			X	X	
Modifiers	X	X	X	X	X	X
Package	X	X	X		X	

Table 4.10: Relevance of tables to error categories

process, to determine which type is imported from where. After that, the table is of lesser relevance to the checking process. The checker focuses on errors in the file under scrutiny, while the import table is mostly concerned with components outside the file.

Chapter 5

Implementation

5.1 Introduction

Once the fact extractor has completed its pass over the source code and delivered all facts that have been found to the database, the next step is to enrich and query this information to perform the actual checks. In particular, since not all information we need can be extracted, the rest has to be gathered here. We have created a prototype static semantic checker to demonstrate how various checks could be implemented. The various errors each have different requirements, so a number of techniques are required to cover all of them. This chapter discusses these techniques, classified according to the errors that are their primary targets. Some checks can be done while the data is inserted into the database, while most can only be done after all data has been inserted. In practice, we found some information is used by multiple checks, so the checker does some preliminary work after the fact extractor has finished its job, but before the main group of checks is executed. Most checks are implemented using a combination of code and SQL, with a few that need SQL queries or contstraints only. While the constraints have an advantage in conceptual elegance, data that does not meet the constraints cannot be inserted at all. This can hinder other checks, that expect a value that is not there. Thus, constraints have been used only sparingly.

5.2 SQL

SQL is a database language designed to express both manipulation and retrieval of information and its structure from relational databases. SQL arose from a language called Sequel that was developed by IBM in the early 1970s[30]. The name SQL is usually taken to be an acronym for Structured Query Language, even if it can do more than just query data. It is used in this thesis to express the queries used in semantics checks. SQL can be divided into three parts, the Data Definition Language, DDL, Data Control Language, DCL and the Data Manipulation Language, DML. The first is used to define and modify tables and their relations. The second is used to control access to the database. It is not described in this thesis. The third, most relevant, part is used to retrieve, insert, manipulate and remove records from the existing tables. The DML offers several types of queries, the two types used here are the select and update queries. The checker prototype also uses some insert queries, which are not discussed here. A select query is used to retrieve records from the database and has the following form:

$$\textbf{SELECT } F_1 \dots F_m \textbf{ FROM } t_1 \dots t_n \textbf{ WHERE } P$$

In SQL, keywords are usually written in uppercase or bolded, while tables and fields are written in lowercase. In the select query, $F_1 \dots F_m$ stands for a list of fields that we want retrieved from the database, $t_1 \dots t_n$ stands for a list of tables these fields belong to, and P stands for a predicate that indicates which records should be returned. P can consist of comparisons, functions or even subqueries, combined by logical operators. If $t_1 \dots t_n$ contains multiple tables, all possible combinations of records are considered for selection. Thus, in order to combine information from multiple tables, we must include a requirement in P that guarantees that only records that belong together can be selected together. If we want to use the same table multiple times, we have to give it a new name with the *as* operator, to avoid ambiguity. Finally, if we want all fields from a given set of tables, we can use the wildcard $*$ instead.

An update query is used to modify records in the database. An update query has the following form:

$$\textbf{UPDATE } t_1 \textbf{ SET } F_1 = V_1 \dots F_n = V_n \textbf{ WHERE } P$$

In the update query, t_1 stands for the table that is to be updated. In the list $F_1 = V_1 \dots F_n = V_n$, F_n stands for the fields that have to be updated and V_n stands for the new values for the fields. These values can be constants, but also the results of subqueries. In the latter case, the subquery must return a single value as its result, otherwise the update fails. The predicate P can be used to limit the updates to a subset of the records in the table, like in the selection query.

5.3 Enriching the database

The preliminary work that has to be done consists mostly of moving some of the raw data in the database to a more convenient place, converting it into a form that is more useful to the checker, and to address some defaults. The defaults are the easiest, and cover properties like the canonical name of local classes, which is based on the package name and the simple name, and the implicit import of the java.lang package that contains basic classes.

One of the most important steps is to add the canonical names of the types used in the program. The canonical names are needed to make sure that the data collected about the types later on is correct. All types in Java have both a canonical name and a simple name. A canonical name is the full name, that should uniquely designate a specific type. A simple name can be derived from a canonical name, is shorter and in practice used more often, but it is only valid in the context of a set of imports. An import can be seen as resolving a simple name to a canonical name, so the name now identifies a unique type. If the checker would use the simple names, as they occur in the program code, there is a possibility that because the checker has a different set of imports than the target program, the simple name is resolved to the wrong canonical name, which results in errors.

So, the canonical names of all types have to be determined. For imports that import only a single type, like for example java.lang.Random, this is easy. The database can be queried directly for the import that matches a given simple type. However, there are also so-called

Field	Value
id	10
name	java.util.Random
class	Random
ondemand	false
scope	1

Table 5.1: Import data example

on-demand imports, that can import multiple types. In that case, no simple name is given, so the only way to discover if a given simple name matches an on-demand import is to use reflection. Reflection cannot be done by the database, so the checker has to query for a list of types that still need names, and then call the reflection functionality that determines if a type with a given name exists using all possible on-demand imports to determine the right canonical name. If a simple type can be matched by multiple on-demand imports, it is ambiguous, which is an error. The easiest way to detect this error is to require that the simple names in the imports table are unique. Once the correct canonical name has been found, it is added to the database. In the example in Listing 3.1 this would result in an entry in the table import with `java.util.Random` as class and Random as name, like in the example in Table 5.1.

The next step is significantly easier. The checker needs information about parameters both as targets of variable references, and to determine which method is called for a given name and arguments. During the fact extraction, only the information for the first part can be extracted easily. In order to simplify the rest of the process, the information for the parameter table for local methods has to be extracted from the variable, fielddec and typeinfo. This can be done with a straightforward insert query, so no intervention by the checker is needed. For non-local methods, the parameter information can be retrieved via reflection, so no special action is necessary.

5.4 Reference errors

References form a major part of all programs and though most of those will be correct, an error is still easily made. The way to approach a reference error depends on the type of the target, for example a method or a variable, that is referenced. The actual discovery of errors in variable and method references will be done during type checking, because the results are needed there.

Variables are referenced by the correct name, which can be matched by the database directly, in the correct scope. Java allows variables to hide some other declarations, so the scoping has to consider the different kinds of scopes and select the closest declaration. The query that is used to link variable references to their declarations is given in Listing 5.1. Some example values are in Table 5.4. The query consists of two main parts, the update part that selects the expressions that need their types determined (i.e. the type field of the record is null), and a select part that selects the appropriate type where possible. The final canonical types are retrieved from the import table, based on the information retrieved from the typeinfo, variable and scope table. The comparison in the first line of the inner `WHERE`

Listing 5.1: Variable reference query

```
1  UPDATE expression SET
2       type=(SELECT DISTINCT import.name
3               FROM import, typeinfo, variable, scope as vscope, scope as escope
4               WHERE import.class=typeinfo.type
5                       AND typeinfo.declaration=variable.declaration
6                       AND expression.text=variable.name
7                       AND vscope.id=variable.scope
8                       AND escope.id=expression.scope
9                       AND scope_contain(vscope.scope,escope.scope))
10  WHERE expression.type IS NULL
```

Expression							
id	**text**	**type**	**kind**	**level**	**pos**	**parent**	**scope**
35	number		name	3	140	34	10

Import				
id	**name**	**class**	**ondemand**	**scope**
6	java.lang.Integer	Integer	false	1

TypeInfo			
id	**type**	**pos**	**declaration**
16	Integer	123	15

Variable				
id	**name**	**pos**	**declaration**	**scope**
18	number	129	15	2

Scope					
id	**lscope**	**rscope**	**level**	**scope**	**parent**
2	2	100	1	(2,100)	1
10	15	78	2	(15,78)	2

Table 5.2: Variable reference database example

clause is used to determine the correct import, and the one in the second line is used to find the right type by the variable. The third line makes sure that the variable name matches the expression text. The next two lines determine the scopes of the expression and the variable declaration. Whether the scopes match is determined by the final comparison made in the selections part, where the appropriate scopes are compared using the scope_contain function. Pure equality is not enough here, because the level field would make correct combinations unequal, while the two-sided nature of the equality would allow incorrect combinations. The function `scope_contain(scope a, scope b)` is defined to be true only when scope b is inside scope a, and not vice versa. The code for this function is given in Listing 5.2 in C code. In the example in Table 5.4, query identify the variable `number` as the reference target and set the type of he expression to `java.lang.Integer`.

Method invocations are the second type of reference. Which method is referenced by a given invocation is determined by the base type, name and the arguments. In particular, the types of the arguments matter. Because of that, methods are treated mostly in the Section 5.5. There, a comparison can be made with the methods used to type operators. As far as reference errors go, any situation where the type assignment process cannot identify the type

Listing 5.2: Scope_contain function

```
1  Datum
2  scope_contain(PG_FUNCTION_ARGS)
3  {
4          Scope    *a = (Scope *) PG_GETARG_POINTER(0);
5          Scope    *b = (Scope *) PG_GETARG_POINTER(1);
6
7          PG_RETURN_BOOL((a->x <= b->x) && (a->y >= b->y));
8  }
```

of a method can be considered a reference error. Either a method is referenced that does not exist, or multiple candidates exist.

Type references are the third type of reference. They occur for example in variable declarations, in class declarations, as super classes and interfaces, and in static method calls. An example is the type of the variable in line 5 of the example program in Table 3.1. The last ones are a bit more troublesome, because unlike most type references, which are resolved and checked in the beginning of the checking process, they are not known at the beginning of the process. Java handles a possible ambiguity between variable names and type names by grouping them under one grammar node, this means they are extracted in the same manner. To determine what a given name refers to, variables are given priority. If no variable with a given name exist, the checker must look for a type definition of that name. When everything else fails, the name must be of a package per the language description.

5.5 Type errors

Finding type errors is one of the main tasks of the checker, and in order to do it, a type needs to be determined for all expressions. In many cases, the type of an expression depends on the types of one or more subexpressions. However, there are number of expression types, like constants and variable references, that can be typed directly. These provide the inputs for the other expressions. For constants, the type can even be added by the fact extractor, since all types of constants have their own nodes in the grammar, so they can each have their own extraction rule. Variables are a little more complicated, because there are possible reference errors to consider. The query used to resolve the types of variables are discussed in the section above.

For expressions that need types of subexpressions to determine their own type, the involvement of the checker is bigger. The various rules about how a given expression depends on the subexpressions is really too complicated to encode in the database as stored procedures or the like. That means that all the database does is provide a list of expressions that still need work in the right order, and the appropriate subexpressions when asked. The checker processes the list of expressions and assigns the appropriate types. Based on the operator encountered, it can decide to query for data on the children, using the query:

```
SELECT * FROM expression WHERE expression.parent=?? ORDER BY expression.pos
```

This query consists of two parts, a static part and a dynamic part. The dynamic part consists of the question marks and is filled in with the identifier of the current expression, based on an earlier query. The static parts take this identifier and retrieve those expressions who have the current expressions as their parent, i.e. the children of this expression. The position is used to determine the left- and righthand child of the operator, which is needed for some expression.

For most operators, this means applying unary or binary numeric promotion, for others, it is only a check if the subexpressions meet the required type and setting the type if those are correct. In the example, the checker will see that the type of the left subexpression of the assignment in line 16 of Table 3.1 is int, and assigns that type to the assignment.

The most complicated part of this process is handling method calls. For most operators, the relation between the "input" types and the output types is quite straightforward and implementation as database functions would be a serious options. Method calls have no obvious relation between input and output, especially if there are multiple possible candidates and the call has to be disambiguated. A further complication is that the reflection facility in Java does not address this directly. It offers a way to directly get information about methods when given a signature, but only for public methods and without taking polymorphism into account. In order to solve this, and to bridge the difference between non-local and local classes, the reflection facilities are used to get all declared methods for each class and insert these into the method table. Because this covers only declared methods, this has to be repeated for all superclasses and so on. This way, the checker has to look in only one table to determine the candidate functions for each call. Based on that list, it can determine the final function by comparing the parameter types to the types of the actual arguments. One consequence of this is that the queries of this part are too fragmented to be described easily, there is too much information retrieved by reflection that would appear to arrive out of the blue.

5.6 Declaration errors

Declaration errors occur when two different declarations of variables, classes or methods are individually correct, but in combination create an ambiguity. This is an error type where the use of a database should provide a notable advantage. In practice, it is not easy to make full use of what the database offers, because the rules are more complicated than what it was designed for. Declaration errors effectively arise from uniqueness constraints, which are also a feature of many database systems. The problem arises when values the database would consider equal are not equal according to the programming language. The first aspect lies in comparing strings. A variable reference can only target variable declarations with the same name, which is essentially a string comparison. The questions is: is the value 'name' equivalent to the value 'Name'?. In programming languages like Java, that are case sensitive, it is not, which corresponds nicely to the string comparison that the database uses in the constraints. Other programming languages like Pascal, however, are case insensitive, so 'name' and 'Name' can be interchanged at will. In that case we need to either create a new data type, with a case-insensitive equality, or somehow convert the appropriate string to lowercase, so the default equality works again.

A second issue is scope. In most programming languages, multiple variables with the same name are no problem, as long as the scopes they are declared in do not conflict. The scope data type described earlier provides the equality that we need. We would probably need a

separate type for each programming language, because of different scope rules.

The third issue lies in comparing method signatures. If the programming language is not polymorphic, this boils down to simply comparing the names of methods and the types of the parameters, in the appropriate order. However, Java is polymorphic. Two methods that have the same name and parameters in the same class are still not allowed, but overriding of inherited methods is. The main consequence is that the database is not suited for this comparison, so it has to be done by the checker. In fact, the checker has to try to imitate the compiler as much as possible here.

5.7 Contradictions

A contradiction occurs when a declaration claims to have a certain property, for example via a modifier, that it does not actually have. Certain modifiers are not allowed to be combined, like abstract and final, because the end results makes no sense. These can be determined by querying the database directly for that combination. Other error types require expression information, like the return types of methods. The type of the returned expression must be a subtype of or identical to the type declared as return type in the method declaration. In the example program, the checker would discover that there are no expressions that return a value in the scope of the method declaration, which is an obvious error.

5.8 Inheritance and interface errors

Inheritance is an important feature of the Java language. Apart from the primordial class java.lang.Object itself classes in Java inherit or override methods from another class. Inheritance effectively combines several classes and interfaces into one, and the result of that combination must be consistent and unambiguous. The base of inheritance is formed by a tree structure, which is a major complication. Accessing the information of the classes individually is no problem, the database takes care of that, but the tree structure has to be handled by the checker in some way. The main complication is that methods from all ancestor classes have to be considered to check if all overriding or hiding methods do so correctly and without creating problems. Furthermore, most of the information required must be gathered via reflection, because usually files will contain only one class, which means that a class that is extended or an interface that is implemented must be non-local.

Thus, determining the details needed to check for inheritance errors requires the full canonical names. After that, the signatures of all methods of the interfaces and superclasses need to be compared to discover if they are correctly implemented or do not override or hide incorrectly. For example, if a method overrides a method from a superclass, the return type of that method must be substitutable for the return type of the overridden method.

5.9 Miscellaneous

As always, there are a number of errors that do not fit in the above categories. As mentioned earlier, their diverse nature makes it hard to discuss implementation details. In the Listing 3.1 program, there is a error related to integer size. The given initial value is to large to fit into an integer. The semantic checker can get a list of literals that are too large by using the

following query:

SELECT $*$ **FROM** expression **WHERE** expression.kind='IntLiteral'
AND str_to_integer(expression.text) > 2147483647

In the example in Table 5.9, the only the second row would be selected, because the value is to large, while to other value is fine. Thus, an error can be reported for that expression.

Another approach to this could be to simple extract the list of literals for each type, and then try to insert the values into a variable of the appropriate type. That would increase the amount of work that has to be done by the checker, but it would be much easier to test for some errors like floating point denormalization.

Expression							
id	**text**	**type**	**kind**	**level**	**pos**	**parent**	**scope**
35	2147483647	java.lang.int	IntLiteral	2	140	34	10
68	2147483648	java.lang.int	IntLiteral	3	289	114	25

Table 5.3: Literal database example

Chapter 6

Prototype results

To asses the theory and feasibility of the design proposed in the previous chapters, we have designed a prototype. There was not enough time to implement all possible checks, so we focused on type checking and other basic errors. In this chapter we present the results of the prototype. Because there are too many errors to discuss individually, we do so in groups of related errors. The full list of errors can be found in Appendix B. In Appendix B, implemented checks are indicated by a +, unimplemented by a - and checks with a different mark have a comment in the appropriate section below.

6.1 Literals

Errors in literals are easy to check. Each type of literal has its own grammar rule, so they can be extracted and gathered easily. For integer and long literals, the value of the literal can be compared to the bound directly. For float literals, it is easier to let the checker test if the value fits in a float variable. Both kinds of checks have been implemented

6.2 Inheritance

Errors concerning inheritance are much harder to check. For each class that is declared in the source file, the ancestor class and interfaces must be determined and checked to discover if an illegal declaration is made. Direct inheritance errors, like trying to extend a final class, can be detected by looking at the base information available for the type. Most of these have been implemented.

For others, like overriding and hiding, method information must be gathered to determine if the methods of the child type conflict with the methods of the parents. Not all of these checks have been implemented, because the process to determine whether two methods can conflict is quite messy, which makes errors in these checks more likely, while also making it harder to test.

6.3 Types

Type checks depend greatly on a correct implementation of the type system. If that system works, many checks boil down to checking if the types of a certain set of expressions meet a

basic criterion. Most of that category of checks have been implemented.

There are also a number of checks that are much more complex. In one case, marked with 0, we have no clear idea what would constitute an incorrect program, which makes it impossible to test or to implement. The check marked by 1 is actually a group of checks, grouped in order to save table space. These checks deal with the typing of all operators. Since some operators have a page or more of rules, we choose not to implement all of them.

6.4 Modifiers

Like most type checks, modifier checks are mostly very similar. Several of these have been implemented, while others have been ignored because of the amount of effort required. An example of the last is the check that abstract or native methods cannot have implementations. This information is not stored in the database, though it could be added if desired. We have chosen not to do so, given that the check would be trivial once the schema have been adapted.

6.5 Enum

The enum construction in Java is essentially a special form of inheritance from the Enum class, that has several special properties. Because of this, many of the errors are inheritance errors, for the special case of the Enum class. Thus, we have chosen not to implement them. The errors that have been implemented are either direct consequences of other errors or trivial.

6.6 Interfaces

Many of the Interface-specific errors deal with generics. Since we have chosen not to consider generics, we cannot implement these errors. An exception is what we considered the most interesting interface check, that requires that a non-abstract class implements all interface methods. This check has been implemented.

6.7 Generics

Again, because we have chosen to ignore generics, these errors cannot be checked. If the checker had to be extended to handle generics, the steps that involve resolving types have to be updated to account for type parameters. In particular, bounds and wildcards must be handled and the full canonical names of the type parameters must be known in order to check if they meet a bound that is present. When that is done, most checks can be derived from those that are already present. A small number of checks require more attention, like the checks to determine what can and cannot refer to the type parameters, because it is more strict than normal scope rules.

6.8 Exceptions

If an error occurs, an exception can be thrown. The exception is passed on until it is caught. This results in three types of exception errors, errors concerning what can be thrown, errors

concerning reporting what can be thrown and errors concerning catching. Of the three types, only the first has been partly implemented, since it follows from the type checking.

6.9 Constructors

Constructors are a special class of methods, that is called when a new object is created. Constructors have some special restrictions concerning where they can be called and what they are allowed to call. We have not paid specific attention to these, because they further complicate the method resolution system, so they have not been implemented.

6.10 Initialization

Java has the requirement that each variable must be initialized before it is used. In addition, there are a number of restrictions on how variables can be initialized, especially if you use a static initializer. We have not implemented any errors in this category, because they deal with context issues that are not handled by the current implementation of the checker.

6.11 Instances & Inner classes

It it possible classes in Java to define classes inside other classes. These classes have special access to the fields and methods the so-called enclosing class(es). Most of the errors in this category deal with restrictions on such references. Other errors include using a reference to a superclass in a place where there is no such class. Because we choose to ignore inner classes, and the specialized nature of the others, none of the errors in this category have been implemented.

6.12 Control flow

Control flow forms a core part of any program. Thus, the control flow structure must be correct for a program to be meaningful. In particular, Java requires that all code is reachable according to a specific algorithm. Because of the complications of that algorithm, we have chosen not to implemented these errors. If they were to be implemented, a modification to the database model, like more detailed statement information, could be useful.

6.13 Ambiguity

References form another core part of any program. If it is unclear what a reference refers to, the compiler doesn't know how to handle that reference. This category contains errors that deal with references that have multiple possible interpretations. A number of these have been implemented.

6.14 Accessibility

Like the previous section, this category deals with incorrect references. In this case, references where the target cannot be found. In a sense, these are closely related to type errors, because

the target of a reference must be known to determine its type. Thus, many of these have been implemented.

6.15 Annotations

Annotations are a recent addition to Java. They are intended to provide information to developers, and have no effect on the execution of the program. Since we choose to ignore annotations, we have not implemented these errors.

Chapter 7

Future work: Templates

7.1 Introduction

As mentioned in the introduction, one of the consideration that lead to this study of the relational approach to static semantic checking was that it would allow the checker to be extended to handle templates easily. Due to time constraints, the focus moved away from templates, but based on the results of the work done on checks for the Java programming language, we can still discuss what kind of modifications would be needed to check templates and what kind of template-specific errors could be tested for. The prototype checker does not feature checks for these errors. In order to simplify this discussion, we will focus on the Repleo[5] template engine. The Repleo engine is based on SDF grammars, just like the fact extractor, which guarantees that the grammar used to describe the templates is directly available for annotation. In the next section, we will give a short description of Repleo. The next section discusses the changes that have to be made to the checker to deal with placeholders. The third section deals with template-specific issues. Finally, the last section contains a discussion of ambiguities, which describes how to deal with a specific problem caused by one of the template constructs.

7.2 Repleo

Repleo[5] is a system to build template engines and related tools based on SDF grammars. The grammar of a given programming language is extended into a grammar for templates of that language by combining it with a set of modules that describe the grammar of the placeholders. Each module describes a generic placeholder, that is instantiated to replace a given node. In effect, we can define for each node in the grammar if and how we want to allow it to be replaced by a placeholder. How the placeholders look depends on the template grammar, but a recommended method is to use <% and %> pairs to mark placeholders clearly. If we take the example Java template grammar provided by Repleo as an example, we could use the (intentionally shoddy) template in Listing 7.1 with the input data in Table 7.1 to get the running example. Tables 7.2 and 7.3 are examples if input data that would create incorrect results. Listings of what the exact result would be can be found in Appendix C. In the example template, we can see two of three types of placeholders: substitution and repetition. The first type, substitution placeholders, are the most common. Each substitution refers to the input data to get their value. The second type of placeholder is the repetition.

They allow the template to be used to create classes with a varying number of fields and corresponding functions. Each repetition must provide a base name, that should indicate the fields that should be repeated. The third main placeholder type is selection, which allows the user of the template to choose between different pieces of code based on input values. Selection is not present in the given example.

Listing 7.1: Template example

```
1  Package Customer// Package declaration
2  import java.util.* // Import declaration
3
4  public class <%record/classname%> {// Substitution
5         private Random randGen = new Random();
6        <%foreach record/field do%> // Repetition
7            private String <%attribute%>; // Substitution
8            int id;
9        <%od%>
10
11       <%foreach record/field do%> // Repetition
12        int <%setname%>(<%type%> <%parameter%>) { // 3 x Substitution
13           id = randGen.nextInt();
14          <%attribute%> = <%parameter%>; // 2 x Substitution
15           return id;
16        }
17
18      <%type%> <%getname%>(int customerID) { // 2 x Substitution
19           if (id==customerID) { // Variable reference
20                       return <%attribute% >; // Variable reference
21           }
22           else {
23                   return null; // Constant expression
24           }
25       <%od%>
26  }
```

7.3 Base language errors

In principle, even a piece of program code without placeholders can be considered a template. It would be a very simple one, with only one possible variation, but it can be parsed according to the template grammar. More practically, in many templates, like in the example template, the amount of base language code will be much larger than the amount of placeholders. Thus, it is very well possible that a template contains errors that are not related to any placeholders and it is desirable to find them. The relational approach meets this requirement, since the template code can be skipped during fact extraction by not adding any annotations to those nodes, or the template facts can be ignored during analysis.

Data structure
record([
classname("CustomerName"),
field ([
attribute("name"),
setname("SetName"),
parameter("customerName"),
getname("GetName"),
type("String")]
)])

Table 7.1: Correct example

Data structure
record([
classname("CustomerData"),
field ([
attribute("name"),
setname("SetName"),
parameter("customerName"),
getname("GetName"),
type("String")]
),
field ([
attribute("age"),
setname("SetAge"),
parameter("customerAge"),
getname("GetAge"),
type("Integer")]
)])

Table 7.2: Repetition example

Data structure
record([
classname("CustomerID"),
field ([
attribute("id"),
setname("SetID"),
parameter("customerID"),
getname("GetID"),
type("String")]
)])

Table 7.3: Duplicate name example

A more interesting situation is where the placeholders do play a role. In the example, the placeholder <%attribute%> in line 14 must refer to a declared variable. In this case, that requirement is met, the repetition starting in line 6 guarantees a variable named <%attribute%> is declared for each field, in this case only one. In order to identify this relation, the checker must know when two placeholders always yield an identical value. If only substitution exists, directly comparing the names of the placeholders could be sufficient. If repetition is added, a more advanced scheme involving normalization is required. Apart from this issue, the situation is not fundamentally different from the non-template case. The scope rules still apply etc. In the same way, if we tried to assign *id* to <%field/attribute%>, it would be incorrect, because *id* is an int and <%field/attribute%> is a String, no matter what name it actually has.

In the example, the type of the variable <%field/attribute%> is declared as String in line 6, but the combination of line 12 and 14 requires it to be of type <%field/type%>, in order to make the assignment correct. Using the input data in Table 7.1, this is no problem because <%field/type%> represents the value String too, so the types match in this case. If we used the data in Table 7.2, the resulting code would have a type error. This demonstrates an issue that often pops up, namely that an certain piece of template is incorrect for nearly all input values, but not all. In this case, the class java.lang.String has been declared final in the standard Java library, so the the values String or java.lang.String are the only valid input values for <%field /attribute%>. Even though this could be an intended effect, we would consider this a clear error. A related error would be when one of the methods contained a static reference to a variable, say *name*. It could be that one of the variables declared by the repetition creates the appropriate target for that reference, but it seems that in most cases this is not what was intended.

A vaguer case is implicit in the first repetition. Each of the <%field/attribute%> values must be different from each other and from the other variables declared, *id* and *randGen*. In contrast to the errors described in the previous paragraph in this case most of the values do not cause any problems. Table 7.3 gives a specific example where it does create an error. If every instance of this construct would be treated as an error, very few templates would be fault-free. On the other hand, if the list of variables is quite long, there are more possibilities for conflicts. Another factor is that variables are not named randomly, which also increases the chances of a duplicate. An obvious compromise is to create a list of restrictions, in some format, that can be used to check if a given set of input data can be used without causing an ambiguity. This is a deviation compared to the other checks suggested here, but it does allow for errors to be detected as early as possible. What form these restrictions could take depends on the template engine, because support of the engine is necessary to guarantee that the restrictions are applied to all input data.

7.4 Template specific errors

In addition to errors that follow from the base language code that is present in any template, there are errors that are specific to placeholders. One example arises in cases like the repetition in line 6 of the example. If the variable *name* was not a placeholder, but a constant, a number of variables with identical names would be declared. This is close to the last error described in the previous section, but the chance of this error causing problems is much larger. If the intention was to create zero or one declaration, a selection placeholder would be a better

choice.

More than the substitution and repetition, selection forms a problem for the checker. A selection creates a situation where there are multiple pieces of code, each with associated facts. This leads to a situation that is akin to ambiguities, which will be discussed below.

Another potential problem lies in the naming system. If the placeholder name <%field/attribute%> was used outside a repetition, it would suggest that well-formed input data has one value for that name. If it is used in a repetition, it suggest there can be multiple values for that name, or none. If both occur in the same program, that could be a reason to emit an error.

7.5 Ambiguities

Ambiguities have been mentioned before in this thesis. In this section, we will discuss syntactic ambiguities, where there are multiple correct parse trees. Many parsers refuse to accept this, and output an error instead of a set of parse tree or even any parse tree. The SGLR parser does output a set of parse trees, called a parse forest. In the parse forest, the various trees are merged as much as possible. Ambiguity nodes represent the location of the ambiguity in the parse tree, with the various alternate trees as subtrees. This gives us the option to consider the various ambiguities in the checker. Each parse tree in the forest has, of course, its own set of corresponding facts. To handle ambiguities, the checker must make sure that facts for different options of the same ambiguity do not interfere with each other. It is, for example, not an error if two variables with the same name occur in two different versions of an ambiguity, because the two will never occur in the same result parse tree.

In order to represent ambiguities in the fact database, we need a way to mark facts in such a way that the checking process is not hindered by irrelevant combinations. We have to consider that ambiguities can be nested, which suggests that we use a path structure or the labelling method used for scopes. The problem is that each ambiguity can have an arbitrary number of alternatives. That means that we have to record the alternative chosen as well. It is not enough to simply determine the subtree-relation, because that does not distinguish between separate ambiguities and separate alternatives of the same ambiguity.

Just because an ambiguity has various syntactical options, does not mean they all are semantically meaningful. Since a semantically incorrect alternative can never be part of a semantically correct parse tree, we can safely choose to eliminate that alternative, which potentially eliminates the ambiguity altogether. There is even the option to do this during the insertion process, so the ambiguous facts are not even inserted in the database.

Chapter 8

Related work

Static semantic checking is done by any modern compiler. In most cases, the theory behind the implementation of the checkers is that of attribute grammars, which originated in a 1967 article by D. Knuth[19]. In this article, Knuth describes how the semantics of a language defined by a context-free grammar can be defined by assigning attributes to non-terminal symbols. In the most basic form of attribute grammars, the values of the attributes are determined by semantic rules based on their direct descendants only. Knuth's main contribution is the introduction of inherited and synthesized attributes. Synthesized attributes are still based on direct descendants, but inherited attributes are based on direct ancestors. While it can be demonstrated that synthesized attributes are sufficient to express all possible semantics for any derivation tree, the resulting semantic rules are often much more complicated. In the paper, Knuth discusses the property of well-definedness. A collection of semantic rules that is well-defined will always lead to definitions for all attributes in all nodes of the derivation tree. Knuth shows that this property can be reduced to that of non-circularity, which can be easily checked for a given collection of semantic rules.

In Chapter 6 of the classic compiler book by Aho, Sethi and Uhlman[1], attribute grammars are used to check the static semantics of a program and determine the semantics information needed for compilation. In particular, type checks, flow-of-control checks for dead code and uniqueness checks for ambiguities are discussed. The type checks are based on type rules that are used to determine the type of each expression. In practice, the main issue is the construction of lists of declared variables or similar information, which are then passed down through the tree where they are used to check for errors.

If we take a more recent book, like the one by A. W. Appel[4] from 1997, we find that Chapter 5 discusses type checks, but no other static semantic checks are discussed in the book. In fact the main subject of the chapter is symbol tables, and how they should be used to handle scoping. These symbol tables are then used to determine the types of variable references.

In general, we can say that attribute grammars combine the distribution of the information throughout the tree with the actual checks. In contrast, in the relational approach we try to separate these aspects more. This closer connection to the parse tree has advantages, because the information in the node is available directly and we do not need to extract it specifically if we need it. On the other, an attribute grammar is always closely tied to a particular grammar and not very modular in nature. If, for example, we wanted to check for template errors specifically, we would add the actual rules to the template nodes only, but

the rest of the nodes still need information to distribute the information throughout the tree. Of course, attribute grammars can be extended in a number of ways. This is the approach taken in JastAdd. The question is which extensions to use. JastAdd uses attribute grammars extended in several different ways, including references [14], circularity[24] and rewriting[9], and there are more options available[23]. Each extension may be able to solve a number of problems, they also have their own problems, under which a lack of support.

In this thesis, we apply the relational method to static semantics. The same approach can be used for related tasks, like software metrics[29] and program slicing[32]. In the first case, a relational approach is used to standardize software metrics and their extraction. Software metrics are numeric values that are calculated based on a piece of software and used to manage, gain understanding or guide improvement that piece. The process used in the article is very similar to the process used here. First, a parser is used to extract facts that are put into a database. These facts are then queried with SQL, which produces the desired metrics. The data model described in the article is focussed on classes, interfaces and methods, which are all stored in a single table called entity. The other tables are filled with the relations between the various entities and their metrics. Concepts like variables or expressions are not extracted. This means that the schema presented is not useful for static semantic checking.

In the second paper, the relational approach is applied to program slicing. A program slice is a part of a program, that has been selected based on some criterion. Usually, this criterion is some instruction in the program, and those parts of the program that affect that instructed (backward slicing) or affected by that instruction (forward slicing). The paper discussed describes a method of extracting slices by first parsing the program using SGLR, then enriching the information in the parse tree using ASF and finally a script written in RScript is used for the final traversal. ASF was discarded for this paper because it was considered to complex to express all facts as tree traversals. This concern is also mentioned in the paper, where it was partly addressed by using a mapping of nodes to a generic set of constructs, so work could be reused easier. Despite that, the complexity of the traversals is still a problem, even though the slicer also ignores types. Because no database is used in this approach, there is no clear relation schema we can compare.

Fact extraction is more than just a part of a relational approach. It is used in areas like reverse engineering, and various programs to do it exist, like Columbus[11], Rigi[31] and SourceML[7]. Articles discussing the theory behind fact extractors are much rarer. One notable one is [22], which discusses various levels of completeness that fact extractors can achieve. The four levels described are source complete, syntax complete, compiler complete and semantically complete. In order, the first requires that the full program, including whitespace, can be constructed based on the fact database, the second that the syntax tree can be recovered, the third that the assembly code can be recovered and the last that the behaviour can be recovered. If we look at our fact extractor, source completeness can be trivially reached, if desired, by extracting the correct facts. The article claims that the other levels are implied by source completion. On the other hand, the fact base created for the semantic checker is not complete on any level, because control structures are extracted only indirectly. Of course, recreating a source program might be important in reverse engineering, it is debatable how relevant it is to other uses of a fact extractor. In a subsequent paper[21] by the same author, fact extraction is defined as a series of transformations, going from source grammar to schema. The goal is to connect the extracted facts closer to the source grammar and to make verification possible. In our case, the fact extractor extracts facts based on annotations in the source grammar. This immediately establishes the desired link between facts and source

grammar. The transformations in the article are not useful to us, because they focus on tree structures, while we want our facts emitted in a list.

A more specific and recent fact-extraction article is [6]. This article describes DeFacto, a fact extractor that also uses the SDF annotations to guide and universal fact extractor. The extracted facts are then processed using RScript to get the required results. Obviously, the annotations used are at a first glance quite similar to the annotations used here. A major difference is that explicitly supports lists, allowing one annotation to extract multiple facts at once. This allows some facts to be extracted in a more convenient way. A limitation is that only source code text can be extracted. This means, for example, that there are no facilities to help with the scope issues identified in Section 4.2.1. In the prototype, DeFacto uses the facilities of Rscript to deal with this. In the article, the DeFacto prototype is compared with ASF+SDF and JastAdd. In both cases, the authors conclude that the DeFacto approach is more succinct and simpler.

In addition, there are several methods for structuring the fact extracted. These are referred to as source models or schemas. Two examples of schemas are the Datrix schema[16] and the Reprise schema[28]. Datrix is a schema intended for C, C++ and similar programming languages. In the article, an E/R schema is discussed for C, C++ and Java. The E/R schema is used to describe the graphs that a legal datrix parser can produce. Similarly, Reprise is also a a graph-based format. The graph nature of these formats makes them more similar to the parse trees that they are based on, but less useful to us because relational databases are not really suited to deal with facts stored in this manner. This is the reason why we chose not to use either format.

Chapter 9

Conclusion

In this paper, we have discussed a relational approach to static semantic checking. The research questions are to discover if a relational approach to static semantic checking could be made sound and complete and would offer advantages over methods based on attribute grammars. We think we have shown that static semantic checking using relational tools is possible. We have yet to discuss two fundamental properties of a static semantic checker: soundness and completeness. In order to make the research possible in the available time, we have chosen to restrict ourselves to a subset of the Java programming language. Even then, the number of errors that can be present in a program in Java is so large that we could not treat them all. So, we can only sensibly talk about the soundness and completeness of a check for a specific error. Obviously, the completeness depends on how the check is implemented. Despite the fact that there is no formal definition available of Java errors, we can compare the checker to the compiler. The fact extractor can extract any information from the source code that the compiler can, and the semantic checker can use reflection to the library information available to the compiler. Thus, it should be possible to define checks in such a way that all errors are found, and only actual errors are found. We have done limited tests by comparing the prototype to the response of a compiler, but without official test cases that evidence is of limited use.

Even if it is possible to make the checks sound and complete, that does not mean that it is easy construct them such that this is the case. This is in effect the third research question, and the answer unfortunately appears to be no. In particular, the database is not really suited to handling tasks related to inheritance and method resolution, where the comparisons in the database are not sufficient. This means we have to use the checker to resolve that, but that complicates matters and means we have to keep converting data from and to the database format. Effectively, we have to reconstruct some of the processes from the Java compiler. The different structure of the checker makes it hard to ensure the process is the same as that of the compiler, and that that the errors are complete and sound. If we look at method resolution, we can see that the information that identifies a method, its signature, is spread of multiple tables: Method and Parameter. That is natural from a extraction perspective or a database perspective, but not from a language perspective. It is clear to us that this structure complicates method resolution, but we have no clear idea how to improve this while maintaining a natural database structure. It must be said that inheritance or method resolution are not straightforward issues in the Java compiler either. Because we did not look at other programming languages, we cannot say what the root cause of the complexity is: the

relational approach, Java, both, or even our understanding of either. At the moment, all we can say is that there is friction between what the database can provide and what the checker needs.

If we compare this to attribute grammars, we can see that the attribute grammars follow the structure of the tree more, like the compiler. That also means that attribute grammars are more tied to the structure of the grammar. One of the goals of the relational approach was to increase the modularity, and we think that is partly achieved. Once all information has been extracted or added to the tree, any check that is desired can be executed in any order. In particular, if any check detects an error, this has little effect on the other checks. This increases the flexibility of the checker. On the other hand, the checks are still quite programming-language specific. The other goal, simplicity, has not been reached. The details of the programming language and the limitations of the database cause a lot of work for the checker, and that results in complexity. Due to this, the prototype checker was a lot more limited than we had hoped.

One consequence of that was that the prototype does not include template support. Based on our experience of implementing the base language checks and the discussion of possible template checks, we can draw some conclusions though. In order to handle templates, first extraction annotations have to be added to the template grammar, based on the annotations of the base language grammar. This leads to alterations of the database schema, to store the template nodes or information to distinguish them from normal nodes. Thirdly, checks will have to be altered to handle placeholders instead direct values. Decisions have to be made about how placeholders and comparisons involving them should be handled. It may be that syntactically different placeholder names actually refer to the same input value, in that case normalization is required. Creating a special type, like the one introduced for scopes, can be a great help here. Finally, there are template-specific checks must be implemented. Ideally, these are independent of the source language and can be reused. In practice, adjustments are likely to be necessary when database schemas differ between base languages.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*, chapter 6. Addison-Wesley, 1986.

[2] Jim Alves-Foss and Fong Shing Lam. Dynamic denotational semantics of java. In *Formal Syntax and Semantics of Java*, pages 201–240, London, UK, 1999. Springer-Verlag.

[3] G. Antoniol, M. Di Penta, G. Masone, and U. Villano. Compiler hacking for source code analysis. *Software Quality Control*, 12(4):383–406, 2004.

[4] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques*, chapter 5. Campridge University Press, 97.

[5] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. Repleo: a syntax-safe template engine. In *Proceedings of the Sixth international conference on Generative programming and component engineering*, pages 25–32, New York, NY, USA, 2007. ACM.

[6] H.J.S. Basten and P. Klint. Defacto: Language-parametric fact extraction from source code. In *Proceedings of the First International Conference on Software Language Engineering*, to be published.

[7] Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. An xml-based lightweight c++ fact extractor. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 134, Washington, DC, USA, 2003. IEEE Computer Society.

[8] A. Van Deursen, J. Heering, H. A. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: a component-based language development environment. In *Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[9] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *Proceedings of European Conference Object-Oriented Programming*, volume 3086 of *LCNS*. Springer-Verlag, 2004.

[10] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[11] Rudolf Ferenc, Árpád Beszédes, Ferenc Magyar, and Tibor Gyimóthy. A short introduction to columbus/can. Technical report, University of Szeged, 2001.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005.

[13] Judith E. Grass and Yihfarn Chen. The c++ information abstractor. In *Proceedings of the USENIX C++ Conference*, pages 265–277, 1990.

[14] Görel Hedin. Reference attribute grammars. *Informatica*, (24):301–317, 2000.

[15] James Hoagland. http://hoagland.org/mkfunctmap.html. Retrieved on 17-11-2008, 1995.

[16] Richard C. Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/r schema for the datrix c/c++/java exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, page 284, Washington, DC, USA, 2000. IEEE Computer Society.

[17] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual*, volume 2b. 1979.

[18] Paul Klint. Rscript tutorial. available at http://homepages.cwi.nl/ paulk/publications/rscript-tutorial.pdf. Retrieved on 17-11-2008, 2005.

[19] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[20] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Softw.*, 21(4):70–77, 2004.

[21] Yuan Lin and Richard C. Holt. Formalizing fact extraction. *Electronic Notes in Theoretical computer Science*, 94:93–102, 2004.

[22] Yuan Lin, Richard C. Holt, and Andrew J. Malton. Completeness of a fact extractor. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 196, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. Extending attribute grammars with collection attributes–evaluation and applications. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 69–80, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Eva Magnusson and Görel Hedin. Circular reference attributed grammars — their evaluation and applications. *Sci. Comput. Program.*, 68(1):21–37, 2007.

[25] Gail C. Murphy, David Notkin, and Erica S. −C. Lan. An empirical study of static call graph extractors. In *ACM Transactions on Software Engineering and Methodology*, pages 90–99, 1998.

[26] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1995.

[27] Java Community Process. http://www.jcp.org/en/resources/tdk. Retrieved on 17-11-2008, 2008.

[28] David S. Rosenblum and Alexander L. Wolf. Representing semantically analyzed c++ code with reprise. In *Proceedings of the USENIX C++ Conference*, pages 119–134, 1991.

[29] Marco Scotto, Alberto Sillitti, Giancarlo Succi, and Tullio Vernazza. A relational approach to software metrics. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1536–1540, New York, NY, USA, 2004. ACM.

[30] Abraham Silberschatz, Henry F. Horth, and S. Sudarshan. *Database System Concepts*, chapter 4. McGraw-Hill Higher Education, 2002.

[31] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: a visualization environment for reverse engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 606–607, New York, NY, USA, 1997. ACM.

[32] Ivan Vankov. Relational approach to program slicing. Master's thesis, University of Amsterdam, 2005.

[33] Jurgen J. Vinju. Uptr: a simple parse tree representation format. In *Software Transformation Systems Workshop*. October 2006.

Appendix A

SDF Grammar of annotations

```
module annotations

imports basic/NatCon
imports basic/StrCon
imports basic/Whitespace
imports basic/IdentifierCon

hiddens
  context-free start-symbols
    Extract

exports
  sorts Extract Type Action Fact Source Data Target

  context-free syntax
    "extractfact(" Type "," Action "," "["{Fact "," }* "]"")" ->
                                                  Extract {cons("ExtractCon")}

    StrCon -> Type {cons("TypeCon")}

    "\"new\"" -> Action {cons("NewCon")}
    "\"use\"" -> Action {cons("UseCon")}

    "\"" "'" StrCon "'"  "->" Target "\""-> Fact {cons("ConstFact")}
    "\"" Source "." Data  "->" Target "\""-> Fact {cons("NodeFact")}

    "$self"    -> Source {cons("SelfSource")}
    "$" NatCon -> Source {cons("ArgsSource")}

    "toString"  -> Data {cons("StrData")}
    "getID"  -> Data {cons("IDData")}
    "getPos" -> Data {cons("PosData")}
    "getFileName" -> Data {cons("FileName")}
```

```
"getBeginLine" -> Data {cons("BeginLine")}
"getEndLine" -> Data {cons("EndLine")}
"getBeginColumn" -> Data {cons("BeginColumn")}
"getEndColumn" -> Data {cons("EndColumn")}
"getOffset" -> Data {cons("Offset")}
"getLength" -> Data {cons("Length")}
"getConstructor" -> Data {cons("Constructor")}
"getLevel" -> Data {cons("Level")}
"getScope" -> Data {cons("Scope")}

IdCon -> Target {cons("Target")}
```

Appendix B

Java compile-time error tables

B.1 Literals

Page	Description	Category	Implemented
23	Integer literal too large	Miscelanuous	+
24	Long literal too large	Miscelanuous	+
26	Float literal too large	Miscelanuous	+
26	Float literal too small	Miscelanuous	+
26	Float literal is small and denormalized	Miscelanuous	+

B.2 Inheritance

Page	Description	Category	Implemented
177	Abstract class not implementable	Inheritance & Interface	-
178, 184	Final class cannot be extended	Inheritance & Interface	+
179	Throwable cannot be extended by generic class	Inheritance & Interface	-
184	Enum cannot be extended	Inheritance & Interface	+
184	Object can have no superclass	Inheritance & Interface	+
184	Cannot extend generic class with incorrect or wildcard type parameters	Inheritance & Interface	-
185	Class cannot depend on itself	Inheritance & Interface	+
217	Cannot override or hide final method	Inheritance & Interface	+
225	Cannot override static method	Inheritance & Interface	+
225	Static method cannot hide instance method	Inheritance & Interface	+
225	Overriding or hiding method must have compatible return type	Inheritance & Interface	+
227	Conflicting override-equivalent methods	Inheritance & Interface	-
228, 229	Conflicting inherited methods	Inheritance & Interface	-
262	Interface cannot depend on itself	Inheritance & Interface	+

B.3 Types

Page	Description	Category	Implemented
62	Intersection type invalid	Type	0
94	Type conversion not possible in this context	Type	+
90	Incompatible classes in class conversion	Type	+
99	Ilegal conversion chain	Type	+
102,103,104	Invalid cast: incompatible classes	Type	+
289, 431	Array element type must be reifiable	Type	+
290, 432	Array index must be indexed by int-values	Type	+
372	If guard must be boolean	Type	+
373	First expression of assertion must be boolean	Type	+
373	Second expression of assertion must not be void	Type	+
377	Switch expression must be char, byte, short, int, Character, Byte, Short, Integer or enum	Type	+
381	While guard must be boolean	Type	+
382	Do guard must be boolean	Type	+
384	For guard must be boolean	Type	+
387	Enhanced for source must be iterable	Type	+
395	Can only synchronize reference type	Type	-
396	Can only catch Throwable or subclass	Type	-
472	void type must be top-level of expression	Type	+
482	Array reference must reference array	Type	-
485-491, 496, 502-515	Operator not applicable	Type	1
491	Illegal cast expression	Type	+

B.4 Modifiers

Page	Description	Category	Implemented
169	Top level type may not be protected, private or static	Declaration	+
175, 197, 214, 260	Duplicate modifier	Declaration	-
181	Inner class can only declare static members that are compile-time constants	Declaration	-
197, 214, 238, 241	Only one of Public, Protected, Private allowed	Contradiction	+
201	Field cannot be final and volatile	Contradiction	+
214	Method cannot be abstract and private, static, final, native, strictfp, synchronized	Contradiction	+
214	Method cannot be native and strictfp	Contradiction	+
216, 472	Attempt to invoke abstract method	Contradiction	+
223	Abstract or native method cannot have implementation	Contradiction	+
223	Non-abstract and non-native method must be implemented	Contradiction	-
228	Method cannot be overriden by method with weaker access	Contradiction	+
241	Enum constructor cannot be public or protected	Contradiction	-
312	Main must be public, static and void	Contradiction	+
362	Local class cannot be public, protected, private or static	Contradiction	-

B.5 Enum

Page	Description	Category	Implemented
176	Abstract method in non-Abstract non-Enum class	Contradiction	+
176, 250	Enum cannot be declared abstract	Contradiction	-
176, 250	Enum must have constants to have abstract method	Declaration	-
176, 250	All Enum constants must implement abstract method	Declaration	-
176, 250	Enum constant may not declare abstract method	Declaration	-
184	Enum cannot be extended	Inheritance	+
249, 424, 425	Enum cannot be instantiated	Type	-
250	Enum cannot be final	Contradiction	-
251	Enum cannot have finalizer	Contradiction	-
252	Enum constructors, instance initializer blocks or instance variable initializer expressions cannot reference a static field of that type that is not a compile-time constant	Reference	-
252	Constructors, instance initializer blocks or instance variable initializer expressions of an enum constant cannot refer to itself or to en enum constant of the same type declared to the right of it	Reference	-
242	Subclasses of Enum cannot invoke constructor of superclass explicitly	Reference	-

B.6 Interfaces

Page	Description	Category	Implemented
186	Cannot implement generic interface with incorrect or wildcard type parameters	Type	-
186	Interface cannot be implemented multiple times	Inheritance & Interface	+
188	Non-abstract class must implement all interface methods	Inheritance & Interface	+
189	Generic interface cannot be implemented multiple times	Inheritance & Interface	+
260	Interface cannot have same simple name as enclosing class or interface	Declaration	-

B.7 Generics

Page	Description	Category	Implemented
50	Only one class or type variable in bound	Declaration	-
50	Type erasures of types in bound must be pairwise different	Inheritance & Interface	-
51	Incorrect number of type parameters	Reference	-
51	Incompatible type parameter type	Type	-
59	Type members cannot be used raw	Type	-
59	Cannot pass type parameter to a non-static type member of a raw type that is not inherited	Reference	-
89	Unchecked warning	Type	-
179	Cannot reference type parameter in static member declaration or static initializer	Reference	-
261	Fields and type members of generic interfaces cannot refer to type parameter	Reference	-
262	Interface cannot depend on self	Inheritance & Interface	-
125, 263, 264	Interface must be compatible with Object	Inheritance & Interface	-
264	Field cannot be redeclared	Declaration	-
264	Member reference ambiguous	Reference	-
265	Initialization expression of an interface field cannot refer to itself or forward	Reference	-
265	Initialization expression of an interface field cannot use super or this unless in anonymous class	Reference	-
267	Interface cannot declare multiple override-equivalent methods	Declaration	-
267	Interface cannot declare final method	Contradiction	-
268	Interface cannot inherit multiple override-equivalent methods unless one is return-substitutable for the others	Inheritance & Interface	-
421	Type variable, parameterized type or array of either have no class literal	Type	-

B.8 Exception

Page	Description	Category	Implemented
202	Exception thrown but not caught or reported	Miscellaneous	-
221, 393	Can only throw Throwable	Type	+
221	Exception must be caught or declared	Miscelanuous	-
222, 226	Overriding method cannot throw incompatible exception	Miscelanuous	-
238	Exceptions thrown by instance initializers must be declared in constructors	Miscelanuous	-
393, 394	Requirements of thrown exception	Miscelanuous	-
398	Cannot catch exception that is not thrown	Miscelanuous	-

B.9 Constructors

Page	Description	Category	Implemented
241	Duplicate constructors	Declaration	-
242	Constructor cannot invoke itself	Reference	-
244	Explicit constructor invocation cannot reference instance variables or methods, this or super	Reference	-
245	Qualified superclass constructor invocation only in non-static inner class	Reference	-
246	Qualified superclass constructor invocation of incorrect type	Reference	-
246	Inner class not member of enclosing class	Reference	-
247	No constructor without arguments or throws clause in superclass	Reference	-
421	This only in instance method, constructor or initializer	Reference	-
425	Cannot instantiate anonymous final class	Contradiction	-
425	Abstract class cannot be instantiated	Contradiction	-

B.10 Initialization

Page	Description	Category	Implemented
71, 397, 527-551	Cannot reassign to final variable	Contradiction	-
199	Blank final must be initialized	Declaration	-
202, 239	Static initializer cannot throw exception or use return	Declaration	-
202	Class variable initializer cannot reference instance variables, this or super	Reference	-
203	Restriction on use of fields during initialization	Reference	-
239	Instance initializer must end normally, cannot return	Declaration	-
367	Variable used before initialization	Reference	-
527-551	Variable not definitely assigned before use	Miscellaneous	-

B.11 Instances & Inner classes

Page	Description	Category	Implemented
422, 439, 441, 471	No such enclosing class	Reference	-
425	Qualified class creation expression must be simple name of non-final inner member of primary	Reference	-
426, 427	Determination of enclosing instance	Reference	-
426, 427	Cannot create instance of non-static local or unqualified inner member class in static context	Reference	-
438, 441	Object cannot use super	Reference	-
473	Illegal target reference	Reference	-

B.12 Control flow

Page	Description	Category	Implemented
223, 392	Method of type void must return normally	Contradiction	-
223, 392	Method of non-void type T must return expression of type T	Contradiction	-
377	Case label must have correct type and non-null	Type	-
378	No duplicate case label or default	Contradiction	-
388	Break without switch, while for or do	Contradiction	-
389	Break with nonexistent label	Reference	-
391	Continue without while, do and for	Contradiction	-
391	Continue must target while, do or for	Contradiction	-
391	Continue with nonexistent label	Reference	-
402	Unreachable code	Miscellaneous	-

B.13 Ambiguity

Page	Description	Category	Implemented
123, 207	Field reference ambiguous	Reference	+
132	Ambiguous type	Reference	+
134	Ambiguous identifier	Reference	-
135, 136	Illegal qualified expression name	Reference	-
137, 444, 447	No such method	Reference	+
137	Illegal qualified method name	Reference	-
154, 167	Duplicate package member or type	Reference	+
162	Duplicate import of distinct type	Reference	+
162, 164	Duplicate import of type and static member	Reference	+
162, 164	Conflict between import and declaration	Declaration	-
196	Duplicate field	Declaration	+
210, 212	Class cannot declare override-equivalent members	Declaration	-
211	Duplicate formal parameter	Declaration	+
362	Duplicate local class declaration	Declaration	-
365, 366, 397	Duplicate local variable declaration	Declaration	+
370	Duplicate label	Contradiction	-
435	Ambiguous field reference	Reference	+
449	Ambiguous method call	Reference	+

B.14 Accessibility

Page	Description	Category	Implemented
44	Type member not accessible	Reference	+
130	Ambiguous name cannot be reclassified	Reference	+
131	Package not in scope	Reference	-
132	Qualified package name not in scope	Reference	-
143	Cannot access non-public field	Reference	-
160	Cannot import from unnamed package	Reference	-
161, 163, 164, 165	Cannot find accessible type to import	Reference	+
184	Class type not accessible	Reference	+
216	Class method cannot reference type variables, instance variables, this and super	Reference	-
238	Static class contains non-static reference	Reference	-
262	Interface not accessible	Reference	+
421	Class not accessible or in scope	Reference	-
424	Class or interface not accessible	Reference	+
435	Cannot access field of non-reference type	Reference	-
435	Can only access accessible fields	Reference	-
441	Cannot statically call interface method	Reference	-
471, 472	Cannot call instance method in static context	Reference	-

B.15 Annotations

Page	Description	Category	Implemented
272	Annotation name already used	Declaration	-
272	Annotation target must match Annotation.Target	Contradiction	-
273	Return type not allowed in annotation	Declaration	-
273	Annotation must be compatible with Object and annotation.Annotation	Declaration	-
274	Annotation of type T cannot contain T	Declaration	-
274, 283	Annotation of incompatible type	Type	-
278	Target type cannot appear twice	Declaration	-
279	Unwarranted override annotation	Miscelleous	-
281	Missing annotation element	Declaration	-
281	Duplicate annotation	Declaration	-
283	Annotation type not accessible	Reference	-
283	Method not present in type	Reference	-

Appendix C

Template Listings

Listing C.1: Duplicate data template listing

```
1  Package Customer// Package declaration
2  import java.util.* // Import declaration
3
4  public class CustomerID{
5         private Random randGen = new Random();
6              private String id;
7              int id;
8
9         int SetID(String customerID) {
10            id = randGen.nextInt();
11            id = customerID;
12            return id;
13        }
14
15       String GetID (int customerID) {
16             if (id==customerID) {
17                           return id;
18             }
19             else {
20                      return null;
21             }
22 }
```

Listing C.2: Repetition data template listing

```
1  Package Customer// Package declaration
2  import java.util.* // Import declaration
3
4  public class CustomerData {
5          private Random randGen = new Random();
6          private String name;
7          int id;
8          private String age;
9          int id;
10
11         int SetName(String customerName) {
12             id = randGen.nextInt();
13             name = customerName;
14             return id;
15         }
16
17         String getName(int customerID) {
18              if (id==customerID) {
19                         return name;
20              }
21              else {
22                      return null;
23              }
24
25         int SetAge(Integer customerAge) {
26             id = randGen.nextInt();
27             age = customerAge;
28             return id;
29         }
30
31         Integer GetAge(int customerID) {
32              if (id==customerID) {
33                         return age;
34              }
35              else {
36                      return null;
37              }
38 }
```