Internal and External
Mechanisms for Extending
Programming Languages

Keren Lenz
Internal and External
Mechanisms for Extending
Programming Languages

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Keren Lenz

Submitted to the Senate of
the Technion — Israel Institute of Technology
Kislev 5773 Haifa November 2012
The research thesis was done under the supervision of Prof. Joseph (Yossi) Gil in the Computer Science Department.

First and foremost, I would like to express my gratitude to my advisor, Prof. Yossi Gil. He has patiently guided me in my journey as a new researcher, providing me with valuable insights and practical guidance, as well as encouragement.

I would like to thank my parents, Hava and Avishay Golz, for years of dedication and devotion and mainly for bestowing me with education as a key principle since early age. Special thanks go to my brother, Omer, for his incredible support and help in parts of this research.

Finally, I am indebted to my beloved husband Oron, for standing beside me every step of the way; and to my Children, Amit, Roy and Ido, for making me happy.

The generous financial support of the Technion, the Harbor fund, and IBM’s PhD Fellowship program is gratefully acknowledged.
CONTENTS

Abstract

1 Introduction
  1.1 The object-relational impedance mismatch
  1.2 Vision
  1.3 Outline
    1.3.1 ARARAT – A relational algebra C++ extension
    1.3.2 Towards a relational algebra JAVA extension

2 Simple and safe SQL queries with C++ templates
  2.1 Integrating Databases and Programming Languages
  2.2 An Overview of ARARAT
    2.2.1 Use of Relational Algebra
  2.3 Preliminaries I: Typing and Templates
    2.3.1 Nominal vs. Structural Typing
    2.3.2 Structural Typing in SQL
    2.3.3 Programming with C++ Templates
  2.4 Preliminaries II: Problem Definition
  2.5 Comparison with Related Work.
  2.6 Relational Algebra Queries
    2.6.1 Composing Query Objects
    2.6.2 Storing Query Objects in Variables
    2.6.3 An Elaborate Example
  2.7 The Ara Little Language
    2.7.1 Syntax
    2.7.2 Semantics
  2.8 System Architecture
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8.1 Combining Compile-time and Runtime Representations</td>
<td>38</td>
</tr>
<tr>
<td>2.8.2 Concepts in RAT</td>
<td>40</td>
</tr>
<tr>
<td>2.8.3 Type Safety of Selection and Projection Expressions</td>
<td>42</td>
</tr>
<tr>
<td>2.8.4 Compiler Extensions</td>
<td>45</td>
</tr>
<tr>
<td>2.9 Structural Typing in ARARAT</td>
<td>46</td>
</tr>
<tr>
<td>2.9.1 Structural Equivalence</td>
<td>46</td>
</tr>
<tr>
<td>2.9.2 Structural Subtyping</td>
<td>48</td>
</tr>
<tr>
<td>2.10 Discussion and Further Research</td>
<td>51</td>
</tr>
<tr>
<td>3 The Use of Overloading in Java Programs</td>
<td>55</td>
</tr>
<tr>
<td>3.1 The Overloading Debate</td>
<td>56</td>
</tr>
<tr>
<td>3.2 Studying The Use of Overloading</td>
<td>57</td>
</tr>
<tr>
<td>3.2.1 On Empirical Study of Programming Languages</td>
<td>59</td>
</tr>
<tr>
<td>3.3 Research Method</td>
<td>60</td>
</tr>
<tr>
<td>3.3.1 Definitions</td>
<td>60</td>
</tr>
<tr>
<td>3.3.2 Data and Automatic Analysis</td>
<td>61</td>
</tr>
<tr>
<td>3.3.3 Sampling and Human Classification</td>
<td>63</td>
</tr>
<tr>
<td>3.4 Taxonomy of Overloading</td>
<td>64</td>
</tr>
<tr>
<td>3.5 Overloading Patterns Catalog</td>
<td>67</td>
</tr>
<tr>
<td>3.5.1 INTRINSIC Overloading</td>
<td>67</td>
</tr>
<tr>
<td>3.5.2 POTENTIAL Overloading</td>
<td>71</td>
</tr>
<tr>
<td>3.5.3 PEER-CALLER Overloading</td>
<td>71</td>
</tr>
<tr>
<td>3.5.4 VISITOR Overloading</td>
<td>72</td>
</tr>
<tr>
<td>3.6 Statistical Analysis of the Experiment</td>
<td>72</td>
</tr>
<tr>
<td>3.6.1 Reliability of Human Classification of Overloading</td>
<td>73</td>
</tr>
<tr>
<td>3.6.2 Binomial proportion confidence interval</td>
<td>74</td>
</tr>
<tr>
<td>3.7 Results</td>
<td>75</td>
</tr>
<tr>
<td>3.7.1 Method Cohorts</td>
<td>75</td>
</tr>
<tr>
<td>3.7.2 Method Pairs</td>
<td>77</td>
</tr>
<tr>
<td>3.7.3 Constructor Cohorts</td>
<td>79</td>
</tr>
<tr>
<td>3.7.4 Constructor Pairs</td>
<td>80</td>
</tr>
<tr>
<td>3.8 Conclusions and Further Research</td>
<td>82</td>
</tr>
<tr>
<td>3.8.1 Summary of Results</td>
<td>82</td>
</tr>
<tr>
<td>3.8.2 Further Research</td>
<td>82</td>
</tr>
</tbody>
</table>
4 Keyword- and Default- Parameters in JAVA

4.1 Solving the hardship of overloading in JAVA 85
4.2 Support for operands and options 89
4.3 A JAVA extension for keyword and default arguments 92
  4.3.1 Methods with Default Arguments 94
  4.3.2 Constructors with Default Arguments 98
4.4 Related Work 100
4.5 Conclusions 103

5 A Microbenchmark Case Study and Lessons Learned 105

5.1 Background: Benchmarking and Micro-benchmarking of Java 106
5.2 Findings 107
5.3 Related Work 108
5.4 Settings of the Benchmark 110
  5.4.1 Hardware and Software Environment 110
  5.4.2 Benchmark Algorithm 110
  5.4.3 Benchmarked Code 111
5.5 No Single Steady State 113
5.6 Examining the No Single Steady State Judgment 117
5.7 Shift of Steady State 120
5.8 The Effect of Prologue Runs 127
5.9 Discussion and Further Research 129

6 Summary 130
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.1</td>
<td>ARA RAT code to construct a simple SQL query</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Nominal typing in PASCAL (a) vs. structural typing in C (b)</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Non-equivalent <code>struct</code> types in C</td>
<td>15</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Computing factorials at compile-time with C++ templates</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Implementing a list of types data structure with templates.</td>
<td>18</td>
</tr>
<tr>
<td>2.4.1</td>
<td>SQLExample</td>
<td>22</td>
</tr>
<tr>
<td>2.5.1</td>
<td>A LINQ code snippet</td>
<td>25</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Writing a simple relational algebra expression in ARA.</td>
<td>29</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Three alternative C++ expressions to compute a query object that, when</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>evaluated, finds the managers of department in divisions located in the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>city of Haifa: using (a) overloaded operators (b) global functions, and (c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>member functions.</td>
<td></td>
</tr>
<tr>
<td>2.6.3</td>
<td>A rewrite of function get_employees using AraRat.</td>
<td>32</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Using ARA RAT to find the names of employees who earn more than their</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>managers.</td>
<td></td>
</tr>
<tr>
<td>2.7.1</td>
<td>The grammar of ARA. Part I: Schema Definition</td>
<td>34</td>
</tr>
<tr>
<td>2.7.2</td>
<td>The grammar of ARA. Part II: statements.</td>
<td>35</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Some semantic checks applied by RAT.</td>
<td>37</td>
</tr>
<tr>
<td>2.8.1</td>
<td>SQL:short1</td>
<td>39</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Figure:S</td>
<td>43</td>
</tr>
<tr>
<td>2.9.1</td>
<td>An inheritance chain composing a record of fields.</td>
<td>47</td>
</tr>
<tr>
<td>2.9.2</td>
<td>An inheritance lattice composing a record of every subset of fields.</td>
<td>48</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Constructor templates emulating structural subtyping</td>
<td>50</td>
</tr>
<tr>
<td>2.10.1</td>
<td>A possible ARA RAT extension supporting dynamic conditions</td>
<td>51</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Different behavior in different languages</td>
<td>59</td>
</tr>
</tbody>
</table>
5.6.2 Distribution of benchmark results, in six additional different computing systems, \( v = 7, r = 20,000, \tau = 100\text{ms} \) . . . . . . . . . . 120

5.7.1 Distribution density of throughput of four auxiliary Java functions, \( v = 3, r = 20,000, \tau = 100\text{ms} \) . . . . . . . . . . . . . . . . 122

5.7.2 Measurement result (throughput ns/ops) vs. session No. of the four auxiliary benchmarked functions. . . . . . . . . . . . . . . . . 124

5.7.3 Measurement result (throughput ns/ops) vs. session No. of the second invocation of benchmark of function listBubbleSort. . . 125

5.7.4 Measurement result (throughput ns/ops) vs. session No. of the four invocations of benchmark of function getCaller on system F (see Table 5.4 and Figure 5.6.2 above), \( r = 20,000, \tau = 100\text{ms} \) . . 126

5.8.1 Throughput vs. \( m \), number of foreign prologue calls, \( r = 1,000, \tau = 100\text{ms} \). 128

5.8.2 Throughput vs. No. of runs after \( m = 50 \) foreign prologue runs, \( \tau = 100\text{ms} \). 128
Abstract

Most object oriented applications that involve persistent data interact with a relational database. There is an inherent gap between object-oriented languages and database access languages, known as “Object-Relational impedance mismatch”. This work strives to bridge this gap in two mainstream object-oriented languages, C++ and JAVA.

We begin by presenting ARARAT, a C++ extension with an augmented relational algebra calculus for the purpose of generating safe SQL queries. All SQL statements constructed by ARARAT are guaranteed to be syntactically correct and type safe, with respect to the database schema. ARARAT allows direct representation of relational algebra expressions as C++ expressions, yet it is implemented without modifications to the host C++ compiler, relying on template meta-programming and operator overloading.

We then describe three studies, conducted in the course of extending JAVA with safe query facilities. First, we present an empirical study on the use and abuse of overloading in JAVA. One interesting contribution of this study is the employment of research techniques traditionally used in social sciences. Second, we extend JAVA method invocation mechanism with named parameters and default values. This feature is realized by modifying the host compiler. Finally, we unveil some of the potential pitfalls encountered while benchmarking small JAVA programs (microbechmarking).
Chapter 1

Introduction

Most contemporary software applications, written in Object-Oriented (OO) languages, handle data originating from two main sources. First, there are runtime values, that is, objects created by the host programming language, often stored in a wide variety of collections which are processed by imperative statements, looping and conditional control structures. Then, there is external data. Historically, the most common external data source was relational databases [23], that have been dominating the area of data processing due to their strong mathematical foundations, simplicity, consistency and ease of use. Still, XML [50] information sources are making their way into the software world at a staggering rate. External data is typically processed by dedicated languages: SQL [55], XPath [22], XQuery [17], etc., and the application language must produce, directly or indirectly, short programs in these typically declarative languages, for the manipulation of external data.

The original purpose of this work was to extend OO languages with new and improved ways of handling external data. In this chapter we present the inherent gap between OO programs and database programs; we then survey various solutions to this gap, followed by our own vision of how it can be best bridged. We continue by describing the work that was done in pursuit of this vision, the caveats we encountered, and the studies that were conducted along the way.
1.1 The object-relational impedance mismatch

The term ‘impedance mismatch’ traditionally refers to difficulties encountered when attempting to connect two systems that have very different conceptual bases. In particular, the term Object-Relational impedance mismatch [83] refers to relational databases and OO languages as systems with their data models, type systems, and programming models as conceptual bases. Mainstream OO languages take different views upon internal and external data. The main characteristics of this mismatch are as follows:

1. The way data is stored and organized. In OO languages the programmer is usually in charge of managing the data representation of collections, which store individual items, using explicit data structures. Databases, on the other hand, free the programmer from worrying about the concerns of physical data representation.

2. Structural vs. Nominal Typing. Records in external storage are necessarily structurally typed—that is, two items are of the same kind if they have the same structure. In contrast, the mainstream programming languages follow nominative typing rules, in which type equality (just as subtyping) is the product of explicit declaration: An operation expecting data of a certain kind cannot be applied to objects of another kind, unless the latter kind was explicitly declared as being compatible with the former.

3. Iterative vs. Declarative Processing. OO collections are usually processed iteratively while relations in database are processed by declarative queries that abstract iterations, by, e.g., SQL or XPath queries. Declarative processing is inherent to database languages, as observed by David Maier [83]: “Whatever the database programming model, it must allow complex, data-intensive operations to be picked out of programs for execution by the storage manager, rather than forcing a record-at-a-time interface.”

These aspects of the mismatch result in difficulties encountered when a relational database is being used by a program written in an OO language. Moreover, to access a relational database from an OO program, the programmer has to write an imperative program whose output is a declarative, database program. This indirect manipulation of external data adds another dimension of difficulty.

Hence, it is no wonder that the field of integrating database languages and programming languages has been studied extensively, and many solutions have
been proposed both by industry and academia. These solutions can be roughly divided into three categories:

1. **Database languages embedded in programming languages**, in which explicit statements in the database language are written inline within the host programming language source code. These statements are then parsed by a preprocessor which replaces them with calls to library APIs. This category includes SQLJ [37], SchemeQL [133] and Embedded SQL for C [75];

2. **Libraries, frameworks and tools which mediate between the database and the programming language**. This category includes Object Relational Mapping (ORM) libraries and tools, which convert data between the incompatible type systems of the database and the programming language, such as ODB \(^1\) and ADO.Net [58]. In addition, there are Call Level Interfaces (CLIs), such as JDBC \(^2\), JDO \(^3\) and EJB [38], which provide mechanisms for accessing and updating databases from programs.

3. **Integrated languages** which combine a general-purpose programming language with a data model. In these languages the manipulation of persistent data is an integral part of the programming language.

   This category includes orthogonal persistence languages, such as OPJ [72], an extension to JAVA [3] which seamlessly integrates the language with the technology for saving and restoring objects and their code to and from stable storage.

   Another way of integrating OO and database languages is through language extensions, which enrich the language with query facilities. The first successful integrated language was probably Pascal/R [110] an extension of PASCAL [138] with relational data types. Pascal/R allows to define relation types, query and alter these using relational operators, and store them in a database.

   The LINQ [88] framework, which integrates query facilities into C# [62], is the most notable integrated language nowadays. In LINQ, data that can be queried include objects in the host language, as well as relational and XML data.

---


\(^2\) [http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/](http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/)

\(^3\) [http://www.oracle.com/technetwork/java/index-jsp-135919.html](http://www.oracle.com/technetwork/java/index-jsp-135919.html)
Java, just as C++ [116], follows the traditional approach of separating persistent data and transient data, leaving the management of persistence to applications or layered mechanisms. The only persistence mechanism offered by Java standard platform is Java object serialization, which allows writing objects to the output stream and reading objects from previously written streams.

1.2 Vision

The goal of this thesis was to extend two of the most commonly used OO languages, C++ and Java, with query facilities.

Our main idea was to use relational algebra operators for data access, allowing direct representation of relational algebra expressions as expressions of the programming language.

This choice offers several advantages: first, it is a thin interface: the six basic operators of the algebra: (i) selection, (ii) projection, (iii) union, (iv) difference, (v) cross product, and (vi) renaming, are complete, in the sense that any other relational algebra operation, such as join and intersection, can be expressed as a sequence of these operations [125]. Second, it is universal: the use of these operators draws on programmers’ prior knowledge and on established unambiguous definitions, as opposed to the many standards of SQL (in the database domain) and the many different ways in which the same logical operation is named in different libraries and collections [108]. Third, it is succinct: the use of terse expressions and operators (as opposed to the verbose, “natural language”, of, e.g., SQL) is more inline with the programming tradition of modern, imperative, expression oriented languages such as C++ and Java.

As both C++ and Java are statically typed languages, a key design criteria was preserving type safety, that is, query expressions will be type checked at compile-time, just like native expressions of the program.

1.3 Outline

1.3.1 ARARAT—A relational algebra C++ extension

The ARARAT system, presented in Chapter 2, extends C++ with an augmented relational algebra calculus for the purpose of generating safe SQL queries. ARARAT achieves a smooth integration of relational algebra operators in the host language,
relying on operator overloading and template programming, without modifying the compiler.

ARARAT addresses all aspects of the Object-Relational impedance mismatch described earlier: It features collection representation transparency, so that the programmer is not aware of the internal organization of the data; it embeds structural type system in C++, relying on the fact that the instantiation of templates is structural; and, it offers declarative operators to process data in a type safe manner.

To appreciate the level of integration of ARARAT in C++, consider the function `get_employees` implemented using ARARAT, in Figure 1.3.1.

Fig. 1.3.1 ARARAT code to construct a simple SQL query

```c
char* get_employees(short dept, char* first) {
#define V(e,EMPLOYEE[FIRST_N,LAST_N])
  // define initial query
  if (first != null)
    e /= (FIRST_N == first);  // apply a selection criteria
  if (dept > 0)
    e /= (DEPTNUM == dept);  // additional selection criteria
  return e.asSQL();
}
```

This function takes two parameters: a short integer `dept` and a string `first`, and returns an SQL statement in string format which evaluates to the set of names of employees, who work in department `dept`, bearing the first name `first`. Each of these conditions narrows the choice only if the respective parameter has a "meaningful" value.

The code that uses ARARAT is short and elegant, and the produced SQL query is guaranteed to be syntactically correct and type safe at compile-time.

1.3.2 Towards a relational algebra JAVA extension

In contrast to ARARAT, extending JAVA with relational algebra calculus cannot be implemented without modifying the host compiler, for two main reasons. First, JAVA generics, unlike C++ templates cannot be used to encode compile-time computations. Second, JAVA does not support operator overloading, which was extensively used in ARARAT to smoothly integrate relational algebra operators within the syntax of C++. Our goal was therefore to extend JAVA with relational algebra types and operators by modifying its compiler.
While designing this language extension, several closely related works were introduced, which greatly reduced the innovative significance of our work.

First, Groovy [77], a dynamic language, based on JAVA and compiles to JVM bytecode, was released. Groovy augments the JAVA collection library with functional constructs, such as filter() and selectAll(), which abstract iteration details from the programmer. In addition, Groovy supports several other features that we intended to include in our work, such operator overloading and lazy evaluation of collections. Thus, Groovy covers many important practical aspects that were planned to be included in our JAVA extension.

Second, at around the same time, several works in the area of statically typed query languages were published. These works include XDuce [63], and other related works (e.g., [131] and [117]), which introduce a statically typed XML processing language, along with its typing rules and type inference algorithms; and a series of works on typing rules for the polymorphic relational algebra ([127], [99] and [19]). The theoretical foundations of these type systems resemble those of our language extension.

In light of these recent results, it became apparent that much of our work on JAVA extension would be technical, and provide little additional research value. Therefore, the goal of defining and implementing a JAVA extension that supports relational algebra type system and operations was not fully met. However, in the course of working on this extension several interesting directions were studied.

The use of overloading in JAVA

The operators of relational algebra are polymorphic. The union operation can take any two relations over the same set of attributes. Moreover, we can join any two relations, regardless of their sets of attributes.

ARARAT makes extensive use of operator overloading to realize this polymorphism. JAVA, on the other hand, does not support operator overloading, since it is considered potentially unsafe, as articulated by James Gosling [119]: “I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++.”

Operator overloading is a critical component of our vision. We did not, however, want to introduce a feature that is likely to be abused. Hence, we conducted a study on the use and abuse of overloading in JAVA. This study is presented in Chapter 3.
For the purpose of this study, we developed a taxonomy of classification of the use of overloading, and applied it to a large JAVA corpus. In this study we found that overloading is extensively used in JAVA; More than 14% of the methods in the corpus were overloaded. In applying the taxonomy to JAVA applications we used sampling and evaluation by human raters. The results of the study show that in at least 60% of the overloading cases, this feature is properly used. In addition, we found that in 35% of the cases overloading is used for emulating default arguments, a mechanism which does not exist in JAVA.

**Keyword and default method arguments**

A relational algebra record is an *unordered* set of attributes. JAVA methods, on the other hand, require their arguments in a specific order. This gap can be bridged by introducing a new method invocation mechanism, *named parameters* [59] (also known as keyword parameters).

In Chapter 4 we present a JAVA extension to support *keyword arguments* and *default arguments*, and further justify the need for this extension based on the results of the empirical study on the use of overloading in JAVA presented in Chapter 3.

**A microbenchmarking experiment**

Introducing structural types inherently poses performance overhead. The dispatch of fields and methods is more complicated compared to nominal type systems, where the location of each field and method is set at compile-time. Structural types were introduced in SCALA [102] \(^4\), and the performance penalty this mechanism introduced was not negligible [35].

In attempt to study the performance penalty of adding structural types to JAVA, we carried out an experiment in which we benchmarked small JAVA functions (microbenchmarking). The findings of this experiment were surprising; even after neutralizing effects of well known perturbing factors such as garbage collection, just-in-time compilation, dynamic loading, and operating system background processes, we encountered inconsistent results.

Chapter 5 presents a case study in which we try to characterize the inconsistencies we encountered.

\(^4\)http://docs.scala-lang.org/style/types.html
Chapter 2

Simple and safe SQL queries with C++ templates

Most large software applications rely on an external relational database for storing and managing persistent data. Typically, such applications interact with the database by first constructing strings that represent SQL statements, and then submitting these for execution by the database engine. The fact that these statements are only checked for correctness at runtime is a source for many potential defects, including type and syntax errors and vulnerability to injection attacks.

The ARARAT system presented here offers a method for dealing with these difficulties by coercing the host C++ compiler to do the necessary checks of the generated strings. A library of templates and preprocessor directives is used to embed in C++ a little language representing an augmented relational algebra formalism. Type checking of this embedded language, carried out by our template library, assures, at compile-time, the correctness and safety of the generated SQL strings. All SQL statements constructed by ARARAT are guaranteed to be syntactically correct, and type safe with respect to the database schema. Moreover, ARARAT statically ensures that the generated statements are immune to all injection attacks.

The standard techniques of “expression templates” and “compile-time symbolic derivation” for compile-time representation of symbolic structures, are enhanced in our system. We demonstrate the support of a type system and a symbol table lookup of the symbolic structure. A key observation of this work is that type equivalence of instantiated nominally typed generics in C++ (as well as other lan-
guages, e.g., JAVA) is structural rather than nominal. This makes it possible to embed the structural type system, characteristic to persistent data management, in the nominal type system of C++.

For some of its advanced features, ARARAT relies on two small extensions to the standard C++ language: the `typeof` pseudo operator and the `__COUNTER__` preprocessor macro.

**Chapter Outline.** The remainder of this chapter is organized as follows. Section 2.1 starts with a description of the problem of integrating a database language and a programming language. Our approach to this problem, the ARARAT system, is presented in Section 2.2. We then continue with two preliminary sections: Section 2.3 discusses the distinction between nominal and structural types and reviews some techniques of C++ template programming, and then Section 2.4 motivates this work by explaining some of the many difficulties raised by the manual process of writing programs that produce SQL statements. Section 2.5 compares ARARAT with other approaches used to integrate database languages within programming languages. The running example presented in Section 2.4 is revisited in Section 2.6, which shows how the same query is expressed in ARA. Section 2.7 then gives a high level description of the entire ARA language. Section 2.8 describes in greater detail the RAT architecture and the template programming techniques used there. Section 2.9 demonstrates how ARARAT supports structural equivalence and structural subtyping in a nominative language. The discussion in Section 2.10 highlights the advantages and limitations of our solution and draws some directions for further research.

### 2.1 Integrating Databases and Programming Languages

Much research effort was invested in the search for the holy grail of seamless integration of database processing with high-level application languages (see, e.g., surveys in [5, 6]). Fruits of this quest include, e.g., the work on Pascal-R [110], a persistent (extended) version [2] of C, integration [28] of databases into SMALLTALK [49], the XJ [60] system integrating XML with JAVA, and many more.

This problem is important since most large software applications make use of a database system. Development resources are invested in the recalcitrant problem of accessing the database from the high level language in which the application is written. The difficulty is that the interaction with a database is carried out in a two step process: First, the application constructs strings—statements in the database
engine control language. Then, these strings are sent for execution by the database engine. Those parts in the application which interact with the database can be thought of as programs that do the non-meager task of composing other programs. An added difficulty with the indirect, meta-programming process is that it is carried out with no aid of supportive, type-safe language environments and CASE tools.

In this chapter, we concentrate on the generation of SQL statements from C++ programs. SQL is still the lingua franca of database processing, with adoptions of the standard [55], in all major database engines including Oracle [76], Microsoft SQL Server [86], MySQL [134], DB2 [90], and many more. Humans may find many advantages in SQL, including readability, well-defined expressive syntax, declarative semantics, flexibility, etc. Most of these advantages are lost however when instead of writing SQL directly, a human is required to write application programs that write SQL. One of the major difficulties in writing such programs, is that the SQL statements they generate are only checked for correctness at runtime. The logic of producing correct SQL strings may be complex. Further, coding errors in the process may invite injection attacks which compromise the safety and integrity of the entire database. These and other difficulties in producing SQL at runtime are discussed in brief below in Section 2.4 and in great detail in the literature (see, e.g., [16, 34]).

2.2 An Overview of ARARAT

ARARAT employs C++ templates to address the common problem of integrating a database language into a programming language. Type checking of this embedded language, carried out by our template library, assures, at compile-time, the correctness and safety of the generated SQL strings. All SQL statements constructed by ARARAT are guaranteed to be syntactically correct, and type safe with respect to the database schema. Moreover, ARARAT statically ensures that the generated statements are immune to all injection attacks.

We describe techniques for embedding little languages within the C++ language, using the template mechanism and without modifications to the compiler. Specifically, we describe the integration of the ARA little language that allows the generation of type safe SQL queries using existing C++ operators. On a broader perspective, our work may be used as a case study by designers of generic mechanisms in future languages. Hopefully, from our experience language designers may gain intuition of the different features that generic mechanisms should offer.
For example, we believe our work makes part of the case for including a mechanism for eliciting the type of an expression, such as the `typeof` operator found in some contemporary compilers, or, the more advanced proposed additions to the language’s standard.

A library of templates is used to embed in C++ a little language [11] representing a relational algebra formalism. In our language, C++ operators replace the mathematical operators of relational algebra, e.g., `+` is union ($\cup$), `/` is selection ($\sigma$). Our language augments relational algebra with abbreviated assignments, e.g., `+=`, and `/=`.

There are two components to the system: a little language, henceforth called ARA, representing augmented relational algebra and a C++ templates and preprocessor directives library, nicknamed RAT, which realizes this language as an embedded language in C++. Thus, RAT is the ARA compiler. Key features of ARA include: (i) tight integration with the host language using template programming, with minimal reliance on external tools and non-standard language extensions, (ii) direct reliance on relational algebra, rather than through SQL. (Some readers may appreciate a measure of elegance in the resulting language.)

2.2.1 Use of Relational Algebra

ARARAT also provides support for the task of defining C++ data structures required for storing the results returned by database queries. The challenge in doing that is significant: the type of a query result is inherently structural in that two queries returning the same set of fields (i.e., named-typed-columns in a relation) must have the same type, regardless of the method by which these fields were collected. In contrast, the types found in mainstream application programming languages are typically nominal: each record type in (say) C++ must be defined and named separately, and once such a definition is made, no other record type is equivalent to that type, even if it has the same set of fields. To bridge the gap between the two systems, ARARAT relies on the observation that the instantiation of generics in C++ (as it is in many other nominative programming languages) is structural. We demonstrate how this unique property of generics can be used to support not only structural equivalence, but also structural subtyping in the nominative type system of C++, that is, we show how template instantiation can be used to define record types in such a way that each such record type is a subtype of all record types which contain a subset of its fields.
Historically, SQL emerged from the seminal work of Codd [23] on relational algebra. SQL makes it possible to encode relational algebra expressions in a syntax which is more readable to humans, and closer to natural language. These advantages are not as important when the queries are written by an application program; we believe that the less verbose relational algebra syntax is more natural and integrates better with imperative languages such as C++ which make extensive use of a rich set of operators. The concise and regular syntax of relational algebra allows modular composition of queries, which is not always possible in SQL.

To appreciate the level of integration of ARARAT with C++, consider the following statement.

```c++
dbcon << (EMPLOYEE[SALARY] / (DEPTNUM == 3));
```

The expression `EMPLOYEE[SALARY]` projects the `EMPLOYEE` relation, selecting only the `SALARY` field. The division operator, `/`, restricts the projected records to those in which `DEPTNUM == 3`. Finally, the left shift operator, `<<`, sends to the stream `dbcon` the resulting SQL statement i.e.,

```sql
select SALARY from EMPLOYEE where DEPTNUM = 3;
```

Moreover, if `q` is a C++ variable storing a query to a database, writing

```c++
TUPLE_T(q) r;
```

defines `r` to be of the type of records that `q` returns.

### 2.3 Preliminaries I: Typing and Templates

#### 2.3.1 Nominal vs. Structural Typing

This section should serve as a reminder of the differences between nominal and structural typing, and explain why we think of the instantiation of C++ templates as structural even though the language is largely nominal. As it turns out, such a mixup between the paradigms, is not C++ specific. Indeed, B. Pierce stated that ‘A few nominal languages have been extended with such “generic” features but the results of such extensions are no longer pure nominal systems, but somewhat complex hybrids of the two approaches.” [106].

Our implementation makes an important use of the observation that template instantiation in C++ is structural. (Still, there are other issues in extending our technique to other programming languages.) Section 2.3.2 below explains why dealing with external data requires structural typing, and provides further intuition of the reasons why SQL is structurally typed. Take note however, that our perspective on
the structural typing properties of C++ templates may be objectional to some, as we shall explain.

A nominal type system, as found, e.g., in PASCAL dictates that a type is equivalent only to itself, and that subtyping is explicit rather than implicit. In contrast, in structural type systems, found, e.g., in HASKELL [71], type compatibility and equivalence are determined based on the structure of the type, i.e., two types are considered compatible if they have identical structure.

Figure 2.3.1 demonstrates the difference between the two. Figure 2.3.1 (a) shows a PASCAL program fragment, in which the two types T1 and T2 are distinct, even though they are both arrays of integers whose indices are in the range 1...100, because they refer to two distinct type definitions. Thus, variable a1 of type T1 is not assignment compatible with variable b1 of type T2, even though the two variables have structurally the same type.

For the same reason, variables a2 and b2 are not assignment compatible. The anonymous type of each of these two vanishes after the variable was declared, and cannot be used after the declaration terminated. (In contrast, variables a3 and b3 refer to the same anonymous type and are therefore assignment compatible with each other.)
Figure 2.3.1 (b) demonstrates structural typing in C: Although variables \( a \) and \( b \) are defined in distinct statements, they are of the same type (an array of 100 elements of type \( \text{int} \)), and are assignment compatible. The same type also receives the names \( T1 \) and \( T2 \) by means of the \texttt{typedef} declarations. But, these type names do not generate new types. All four variables defined in this short code excerpt, \( a, b, c1 \) and \( c2 \), have the same exact type and are assignment compatible.

Unlike arrays, pointers, and other C type constructors which are structurally typed, equivalence of aggregate type constructors, that is \texttt{struct} and \texttt{union} types in C is determined nominally, as demonstrated in Figure 2.3.2. Each of the variables \( p1 \) and \( p2 \) in the figure refers to its own anonymous type which is discarded after the variables are defined.

---

**Fig. 2.3.2** Non-equivalent \texttt{struct} types in C

\[
\begin{array}{ll}
\text{struct} & \text{struct} \\
\{ & \\
\text{int} & \text{int} \\
\text{id;} & \text{id;} \\
\text{char*} & \text{char*} \\
\text{name;} & \text{name;} \\
\} & \} \\
p1; & p2;
\end{array}
\]

Nominal typing of aggregate types carries through from C to C++’s \texttt{class} type constructor. Moreover, \textit{subtyping} of class types is nominal rather than structural; one class is a subtype of another only if it is declared as such. Such a declaration also dictates that the set of members of the subtype is a super set of the set of members of the supertype. In contrast, in structural typing system, a type is a super type of another if every member found in the former is also found in the latter.

Although type names play a less crucial role in structural type systems, such systems usually allow naming types; for example, the following defines two named types whose equivalence is structural.

\[
\begin{align*}
\text{typedef void} & \texttt{(*g)} (\text{int} \text{ a, int b}); \\
\text{typedef void} & \texttt{(*f)} (\text{int} \text{ i, int j});
\end{align*}
\]

The structural equivalence implies that a value of type \( f \) may be assigned to a variable of type \( g \).

### 2.3.2 Structural Typing in SQL

The many advantages of nominal type systems (see, e.g., [84, 106] for a discussion of the relative merits of the two approaches), make these the rule in mainstream programming languages, including \texttt{JAVA}, \texttt{C\#}, \texttt{SMALLTALK}, and \texttt{C++} (aggregate
types). However, there is one crucial point in which nominal type systems fail, that is, the interaction with external data: Data stored on permanent storage, data received from communication lines, and all other external data cannot be nominally typed, since all nominal types declared within a program vanish when the program terminates. Nominal types defined in distinct programs are necessarily distinct, and therefore data serving for program communication must be structurally typed.

As SQL deals with permanent data, it must therefore be structurally typed. Other reasons for adopting this typing rule include: the reliance on relational algebra, which is structural in its essence, and the need to apply a union of tables of the same structure generated by different sequence of operations.

A fundamental rule of SQL is that two tables are mutually compatible if they have the same set of fields. This rule applies not only to the basic relations as they are found in the database but also to interim relations obtained in the course of computing queries: a union (say) of two subqueries is possible if they both contain the same set of fields. The difficulty that ARARAT faces is in using the nominal type system, with types particular to the programs they are defined in, for manipulating external data which was not created using these types.

What made ARARAT possible is the observation that template instantiations (just as arrays) are structurally typed in the following sense. If \texttt{Stack} is a C++ \texttt{template} taking a single type parameter, then variables \texttt{a} and \texttt{b} defined by

\begin{verbatim}
Stack<int> a;
Stack<int> b;
\end{verbatim}

are compatible.

As shall be explained below, the compatibility of variables \texttt{x} and \texttt{y} defined by

\begin{verbatim}
DEF_V(x, DIVISION*DEPARTMENT);
// Variable \texttt{x} has the type of the join of two relations.
DEF_V(y, DEPARTMENT*DIVISION);
// Variable \texttt{y} has the type of the same join, but in opposite order.
\end{verbatim}

is achieved by using two structurally identical sequences of template instantiations, that generate the same type. (The macro \texttt{DEF_V(v, E)} defines a variable \texttt{v} whose type is the same as that of relational expression \texttt{E}).

\textbf{The Single Instantiation Rule} Consider again an example such as

\begin{verbatim}
Stack<int> a;
Stack<int> b;
\end{verbatim}

\begin{verbatim}
The fact that \( a \) and \( b \) are of the same type can be explained not by structural equivalence of template instantiation, but rather by what is known as the “single instantiation rule” (which may be thought of as lazy evaluation of type functions). From this perspective, only the first occurrence of \( \text{Stack<int>} \) leads to the generation of a new type. All subsequent occurrences refer to exactly that type. Further, the fact that \( a \) and \( b \) are of the same type is explained by the compiler fetching the mangled name of that type from its symbol table, rather than by taking its structure into account. Thus, by this perspective we do not have structural typing at all in template instantiation, but rather a single type, with a single name, which is referenced from different code locations. In the example above, this name is \( \text{Stack<int>}, \) or whatever mangled name the compiler chooses to call it.

Issues with this alternative perspective may be the perceived promotion of mangling, an internal compilation strategy, into the language type system, and the fact that type equivalence occurs even when there is no instantiation at all, as in

\[
\text{Stack<int>} \ast p; \quad \text{// No instantiation of Stack<int>} \text{ is required here,}\n\]
\[
\text{Stack<int>} \ast q; \quad \text{// neither here.}\n\]
\[
\text{// ...}\n\]
\[
p = q; \quad \text{// and q are compatible}\n\]

Another concern is that identical instantiations of a template in distinct compilation units may still generate distinct, yet identical types in some compilers.\(^1\)

Note that structural typing of template instantiation is in accordance with C++’s “single instantiation rule”. Without structural typing, multiple instantiations of a class template with the same arguments, would have always resulted in multiple copies of that class in the generated binary. If this was the case, the acute code bloat problem due to template instantiation, could not be solved by clever techniques such as those described by B. Stroustrup in chapters 15.6.3 and 15.10.4 of the “The Design and Evolution of C++” [115]. Still, structural equivalence of generic instantiation is found in other nominally typed languages in which the problem of code bloat is minimal (e.g., C\#) or non-existent (e.g., JAVA).

### 2.3.3 Programming with C++ Templates

C++ templates were initially designed for generic data structures and algorithms, as used in STL [107]. The community, however, has observed that this language mechanism is powerful enough to serve many other tasks. This section reviews

---

\(^1\)See for example, Marshall Cline famous C++ FAQ, questions 35.13, 35.15.
some of the techniques of template programming, giving the intuition why templates can be used to encode all algorithms, and how this encoding may be achieved. Readers who are familiar with template programming techniques may skip forward to Section 2.4.

Figure 2.3.3 shows a simple example which, relying on integer constant parameters to templates, computes factorials at compile-time.

**Fig. 2.3.3** Computing factorials at compile-time with C++ templates

```cpp
template<int N>
class factorial {
   public: enum {result = N*factorial<N-1>::result};
};

class factorial<1> {
   public: enum {result = 1};
};
```

The first template in the figure makes the recursive definition. The *template specialization* (lines 6 through 8) is the recursion base. The figure thus demonstrates two important techniques of template programming: using recursive calls for iteration and specialization for conditions. Together, the two techniques make C++ templates (unlike JAVA generics) Turing complete (see, e.g., [18] and Gutterman’s [56] emulation of two-stacks Turing machine with templates). Turing completeness means that there exists a C++ templates implementation of any algorithm that can be implemented in (say) C++. Deriving this implementation is sometimes difficult; the literature offers many techniques for making this task easier.

A basic technique is of using templates to represent data structures. Figure 2.3.4 for example shows how *typedef* instructions inside a class make it possible for one type to store what can be thought of as “pointer” to another type.

**Fig. 2.3.4** Implementing a list of types data structure with templates.

```cpp
template<typename First, typename Rest>
class Cons {
   // Generate a list whose first element is First while the remaining elements are Rest.
   public:
      typedef First CAR; // Use a Lisp like notation,
      typedef Rest CDR; // also for the CDR component.
};
```
The template Cons in the figure demonstrates how LISP-like lists can be implemented. Recursive, LISP-like processing of lists, lists-of-lists, is common in the field of template programming, with the recursion being realized by template specialization, (much as in Figure 2.3.3). The generalization of lists to trees of types, lookup-tables and other advanced data structures, is not difficult and has become part of the tools of the trade. There is even a library of data structures and algorithms, Boost MPL,\(^1\) which manipulates types at compile-time much like STL manipulates data at runtime. Boost even includes an implementation of unnamed functions, in the form of \(\lambda\)-expressions [68].

An important technique, to which we will refer later, is the standard encoding of symbolic expressions as types, e.g., for symbolic derivation (SEMT [45]) or for emitting efficient code after gathering all operations applicable to a large vector [129]. Three principles are applied by such an encoding:

1. Literals are encoded either as simple types, or as templates receiving the literal value.

   ```cpp
class EMPTY_SET {
    // the \(\emptyset\) literal, used, e.g., in set-theoretical symbolic expressions.
};
template<int N> class NUMBER {
    // an integral value, used, e.g., in arithmetical symbolic expressions.
    public: enum {value = N};
};
template<char C> class VARIABLE {
    // a symbolic variable, denoted by character C
    public: enum {name = C};
};
```

2. An \(n\)-ary operator is encoded by a template taking \(n\)-arguments. For example, binary addition operator may be encoded by

   ```cpp
template<typename L, typename R> class PLUS {
    public: typedef L left; typedef R right;
};
```

3. The type encoding of a non-atomic symbolic expression is defined recursively as follows. Apply the template of the top most operator, to the types of the arguments of this operator. So, the type encoding of \(e_1 + e_2\), is

\[\text{PLUS}(<\text{NUMBER}(<\text{INT}>), <\text{NUMBER}(<\text{INT}>))}\]

PLUS\langle T_1, T_2 \rangle \text{ where } T_1 \text{ is type encoding of } e_1 \text{ and } T_2 \text{ is the type encoding of } e_2.

At a later point in time, but still at compile-time, this type structure can be processed, often by a recursive traversal, to generate other compile-time structures, or to generate actual code.

Some of the diverse applications in which template programming is used include dimensional analysis [126], a solution of the co-variance problem [118], and even for implementing a framework for aspect-oriented programming [143]. Czarnecki and Eisenecker, in their work on application of template programming [32], marked the emergence of a research trend on generative programming [9, 48, 73, 105]. RAT uses many of the techniques developed in this research, including methods for different overloading of the same operator for different sets of types, traits [97], etc.

Most of the ARA's work is carried out by the C++ compiler itself, which is driven by template programming to make sure that the compiled program will generate only correct and safe SQL. Our implementation started from the established techniques of template programming [1, 32], and extended these to meet the challenges of realizing a little language. For example, the techniques for producing (relatively) meaningful and short compilation errors in response for errors in use of the template library are borrowed from [87, 111].

RAT representation of relational algebra expressions is hybrid, straddling the compile-time and runtime worlds. The C++ type of an ARA expression reflects the schema of the associated query, allowing type checking the query at compile-time, while the actual evaluation procedure, that is, the Abstract Syntax Tree of the query, is represented as a runtime value. This hybrid representation allows repeated assignment of different queries that share the same result schema, to a single C++ variable, whose type encodes that schema. The compile-time encoding of the result schema is used for generation of a data type for the query result, which represents a single row in the result relation.

This work shows that the C++ template mechanism is rich enough to support the complex algorithms required for this task. Note that there are several previous examples of embedding a little language in C++, including, e.g., the Boost Spirit library\textsuperscript{1} in which BNF syntax is embedded within C++, while the compiler, as part of the compilation process generates an LL parser for the given syntax. Our system

\textsuperscript{1}spirit.sourceforge.net/
differs from this (and other such examples) in that the type system of Boost Spirit is degenerate in the sense that it has a fixed small set of types: grammars, actors, closures, etc. All symbols in a BNF belong to a single type. In contrast, ARARAT features a multi-sorted type system, allowing two queries to be combined only if they are compatible.

A specific challenge in this management is that the type system of queries is structural rather than nominal. The observation that made it possible to meet this challenge is that instantiations of generic classes obeys structural equivalence rules, even in nominally typed languages. We further demonstrate that non-trivial symbol table management, can be carried out as part of template based computation.

### 2.4 Preliminaries II: Problem Definition

This section motivates our work by demonstrating the intricacies of the existing string based database access from C++.

Table 2.1 introduces a database schema that will be used as a running example throughout the chapter.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE</td>
<td>ID (int), DEPTNUM(smallint), FIRST_N(varchar), LAST_N(varchar), SALARY(double), LOCATION(varchar)</td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>ID(smallint), MANAGER(int), DESC(varchar), DIVNUM(smallint)</td>
</tr>
<tr>
<td>DIVISION</td>
<td>DIVNUM(smallint), CITY(varchar)</td>
</tr>
</tbody>
</table>

There are three relations in this database, representing employees, departments and divisions. Field DEPTNUM is used in relation EMPLOYEE as a foreign key for referencing relation DEPARTMENT, i.e., a typical join operation of EMPLOYEE and DEPARTMENT will involve renaming of field ID in DEPARTMENT to DEPTNUM. For simplicity, we used field DIVNUM both as a primary key in the third relation and a foreign key in the second one. Also, field MANAGER in DEPARTMENT denotes the employee number of the manager of this organizational unit.

Observe that the field named ID has type int in EMPLOYEE and type smallint in DEPARTMENT. The ARARAT type manager supports this.
Figure 2.4.1 shows a simple C++ function that might be used in an application that uses the database described in Table 2.1.

**Fig. 2.4.1** Function returning an erroneous SQL query string.

```cpp
const char* get_employees(int dept, char* first) {
    bool first_cond = true;
    string& s = *new string("SELECT FIRST_N, LAST_N FROM EMPLOYES ");
    if (dept > 0) { // valid department number
        s.append("WHERE DEPTNUM = ");
        s.append(itoa(dept));
        s.append(" ");
        first_cond = false;
    }
    if (first == null)
        return s.c_str();
    if (first_cond)
        s.append("WHERE ");
    else
        s.append("AND");
    s.append("FIRST_N= ");
    s.append(first);
    s.append(" ");
    return s.c_str();
}
```

Function `get_employees` receives an integer `dept` and a string `first`, and returns an SQL statement in string format that evaluates to the set of all employees who work in department `dept` bearing the first name `first`.

The function presented in the figure also takes care of the special cases that `first` is `null` or `dept` is not a valid department number.

Note that this is only a small example. In a typical database application, there are many similar functions that are often much more complex. Furthermore, every change to the application functionality will necessarily result in change to those functions’ logic.

Further scrutiny of the body of `get_employees` reveals more errors and problems:

1. **Misspelled name.** A typo lurks in line 4, by which `EMPLOYES` is used (in-
stead of EMPLOYEE) for the table name. This typo will go undetected by the compiler.

2. **Syntax error.** There is no space after the AND keyword in line 18, and thus, if both parameters are non-null, the produced statement is syntactically incorrect.

3. **Type mismatch.** The SQL data type of DEPTNUM column is `smallint`, while the corresponding input parameter is of type `int`. Such errors might result in unexpected behavior. The source of this kind of problem, which is sometimes called in the literature *impedance mismatch* [140], an inherent problem of integrating database engines and programming languages. One aspect of the difficulty is that the type systems of the underlying programming language (C++ in this case) and the database language (SQL) are distinct—one may even say, foreign to each other—and thus very little checking can be done at the junction of the two languages.

4. **Security vulnerability.** The code is vulnerable to SQL script injection attacks [64] as, a malicious user can construct a value of the first parameter that executes unexpected statement that harms the database.

5. **Code coverage.** Every SQL statement in the code should be executed during testing to validate its correctness. Providing a full code coverage of all execution paths is a demanding task.

6. **Maintenance cost.** The database structure must be kept in sync with the source code. Changes to the database schema might require changes to the SQL queries in the application, which makes the maintenance of the code base harder.

Which of these problems are solved by ARARAT? First, name misspelling will result in compile-time error (we do assume that the database schema was fed correctly into the system). Other syntax errors will be caught even without this assumption. Also, RAT takes care of the correct conversion of C++ literals and values to SQL.

Strings generated by ARARAT are immune to injection attacks. RAT defense against injection attack is twofold: (i) For non-string types, RAT and the C++ type system prevent such attacks. Consider, e.g., a query generated by the following simple C++ code:

23
char *makeSelectUserQuery(char *name, char *pin) {
    char *res = (char*)malloc(1000+strlen(name)+strlen(pin));
    sprintf(res, "select * from users where name='%s' and pin=%s", name, pin);
    return res;
}

This simple function is vulnerable to attacks in which a malicious user supplies
the string "1 or 1=1" (without the quotes) into variable pin. This injected value
becomes part of the generated SQL statement,

\[
\text{select * from users where name='admin' and pin=1 or 1=1}
\]

i.e., the filter becomes a tautology. These kinds of attacks are not possible with
ARARAT, since pin must be an integer. (ii) RAT's protection against injections
into string variables, e.g., into variable name in the above example, is carried out
at runtime. As we shall see below (Figure 2.7.2), ARA allows user string variables
only as part of scalar expressions, used for either making a selection condition,
or for defining new field values. Therefore, RAT's runtime can, and indeed does,
validate the contents of string variables, escaping as necessary all characters that
might interfere with the correct structure of the generated SQL statement.

The maintenance cost is minimized by ARARAT. A change to the scheme
requires a re-run of the DB2ARA tool (or a manual production of its output), but
after this is done, mismatches of the generated SQL to the schema are flagged with
compilation errors.

2.5 Comparison with Related Work.

A number of constructs of SQL address some of the difficulties of programmatical
composition: a systematic use of prepared statements adds a layer of safety at the
cost of excluding dynamic composition of statements; command parameters defend
against injection attacks, but not against syntax errors arising from dynamical com-
position of statements. Our focus here is in more than that: a closer integration, as
seamless as possible, of SQL with the host language, an issue which has engaged
the research community for at least a decade.

A first approach to this problem is in processing of embedded SQL statements
by a dedicated external preprocessor as done in, e.g., SQLJ [37], SchemeQL [133]
and Embedded SQL for C.\(^1\) Advantages are in preserving flexibility and power
of both languages. In contrast, this approach does not support very well dynamic
generation of statements—all possible SQL statements must be available to the
preprocessor for inspection.

Second, it is possible to use a software library, as done in the SQL DOM sys-

tem [85] whose structure reflects the constraints which the database schema im-
oposes, and allows production of correct SQL statements only. Such systems trade
expressive power for usability and require the user to use a large schema-specific
library, with, e.g., a class for each field, a class for each condition on each field,
etc.

Third, in LINQ [88], Xen [89] and the work of Van Wyk, Krishnan, Lijesh,
Bodin and Johnson [141] the host language compiler is modified to support SQL-
(or XML-)like syntax. The advantage is the quality of integration, and the minimal
learning curve. An important variation of this approach is offered by Cook and
Rai’s work entitled Safe Query Objects [27] which relies on the reflective features
of OpenJava [122] to achieve such syntactical changes and to encode constraints
imposed by the schema. Concerns with this approach include portability, expres-
sive power, and quality of integration of the foreign syntax with other language fea-
tures: It is never possible to embed in full a foreign syntax in the host language, the
result being that SQL is changed in many subtle and sometimes not so subtle ways,
whereby generating a yet new SQL variant. Consider for example, the LINQ code
snippet in Figure 2.5.1 which is drawn from Microsoft documentation. Despite us-

\begin{verbatim}
 customers.Select(c => new {
   c.Name,
   TotalOrders = c.Orders.
      Where(o => o.OrderDate.Year == year).
       Sum(o => o.Total)
 });
\end{verbatim}

\begin{footnote}
\end{footnote}
resemblance to SQL statements, this resemblance may be misleading, since not all SQL statements have an equivalent LINQ representation; the programmer thus need to learn a new variant of SQL in order to use LINQ. In contrast, ARAARAT is isomorphic to relational algebra, the only differences being operator names and ARAARAT’s support for assignment.

Finally, there is the approach of using an external static analyzer that checks that the program only produces correct SQL statements [21, 52, 57]. Again, usability is limited by the necessity of invoking an external tool. Also, unlike the other approaches, this approach does not simplify the engineering work of coding the production of SQL queries. Since the analysis is only partial, it does produce false alarms.

In comparison to these, and thanks to the template-based implementation, in ARAARAT we achieve a high-level of integration with the host language without using a software library tailored to the database. In this respect, ARAARAT is somewhat similar to the HaskellDB [79] system, an SQL library for Haskell. In HaskellDB, just like in ARAARAT, columns of a query’s result are carried in the type of the query, and therefore the support of dynamic queries is limited to changes to the conditions of the where clause. However, unlike HaskellDB, by relying on C++ ARAARAT is more accessible to application programmers.

Another issue common to all approaches is the integration of the SQL type system with that of the host language. Given a description of a database schema, which can be obtained using the DB2ARA tool or manually, ARAARAT automatically defines a class for each possible fields combination, with a fixed mapping of SQL types to C++ types. Such a class can be used for data retrieval and manipulation. Also, ARAARAT allows for a manipulation of SQL query objects at runtime—that is, one can construct a query object, and then refine the query, without actually executing it (see Figure 2.6.3 below for an example).

Admittedly, just like many other systems mentioned above, ARAARAT is language specific. Also, just like other research systems it is not a full blown database solution. The current implementation demonstrates the ideas with a safe generation of queries. ARAARAT extensions for integration of query execution are left for future research, or for a commercialized implementation. We further admit that unlike LINQ, ARAARAT is not industrial-strength, e.g., addition of user-defined functions to the core library is not as smooth as it ought to be, the support of vendor specific SQL dialects is not optimized, update operations are left for further research, etc.

26
We should also mention C++ libraries, including, e.g., *Ultimate++*\(^2\), *Terimber*\(^3\), *CQL*\(^4\), and *SourcePro©-DB*\(^5\) which provide an extensive set of functions that can be used for the production of SQL. The support for the generation of SQL statements using these libraries ranges from none to elaborate mechanisms found in *Ultimate++* allowing partial emulation of the SQL syntax with C++ functions calls, as in

```c++
SqlSet borrowed =
    Select(BOOK_ID).
    From(BORROW_RECORD).
    Where(IsNull(RETURNED));
```

However, none of these libraries apply static type checking to the produced SQL statements. On the other hand, these libraries excel in many aspects of database management which *ARA* neglects, including the managements of connections, passwords, transactions, etc.

### 2.6 Relational Algebra Queries

An *ARA* programmer wishing to execute a database query must create first a *query object*, encoding both the specification of the query and the schema of its result. The query object can then be used for executing the query, defining variables for storing its result, and for other purposes.

Just like all C++ objects, query objects have two main properties:

1. **Type.** The type of a query object encodes in it the schema of the result of the query. The *RAT* library computes at compile-time the data type of each tuple in the result. If two distinct queries return the same set of fields, then the two query objects representing these queries will encode in them the same result type.

2. **Content.** The content of a query object, which is computed at runtime by the *RAT* library, is an abstract encoding of the procedure by which the query might be executed. All query objects have an `asSQL()` method which translates this abstract encoding into a string with the SQL statement which can

\(^2\)http://www.ultimatepp.org
\(^3\)http://www.terimber.com
\(^4\)http://www.cql.com
\(^5\)http://www.roguewave.com/sourcepro/sourcepro.cfm
be used to execute the query. Also, query objects have a conversion operator to string, that invokes \texttt{asSQL()}, so query objects can be used anywhere a value of type \texttt{const char*} can be used.

The DB2ARA pre-processor tool generates a \textit{primitive} query object for each of the relations in the input database. The content of each primitive query object is an encoding of the pseudo-code instruction: “return all fields of the relation”. In the running example, header file \texttt{employees.h} defines three such primitives: \texttt{EMPLOYEE}, \texttt{DEPARTMENT} and \texttt{DIVISION}. Thus, the C++ expression \texttt{EMPLOYEE.asSQL()} (for example) will return the SQL statement,

\begin{verbatim}
  select * from EMPLOYEE;
\end{verbatim}

A programmer may compose more interesting query objects out of the primitives. For this composition, the RAT library provides a number of functions and overloaded operators. Each of the relational algebra operators has a C++ counterpart. It is thus possible to write expressions of relational algebra, almost verbatim, in C++.

### 2.6.1 Composing Query Objects

Figure 2.6.1 shows a C++ program demonstrating how a compound query object is put together in ARA. This query object is then converted to an SQL statement ready for execution.

In lines 10 through 15 of this figure, a compound query object is generated in two steps:

- First (line 10), the expression
  \begin{verbatim}
  EMPLOYEE / (DEPTNUM > 3 && SALARY <= 100000 )
  \end{verbatim}

  evaluates to the query object representing a selection of these tuples of relation \texttt{EMPLOYEE} in which \texttt{DEPTNUM} is greater than 3 and \texttt{SALARY} is no greater than $100,000.

  The syntax is straightforward: the selection criterion is written as a C++ boolean expression, and operator / is used for applying this criterion to \texttt{EMPLOYEE}.

- In lines 11 through 15 an array access operation, i.e., \texttt{operator []}, is employed to project these tuples into a relation schema consisting of four fields:
Writing a simple relational algebra expression in ARA.

```c
// Global RAT declarations and macros
#include "rat"  // Global RAT declarations and macros
#include "employees.h"  // Primitive query objects and scheme of the EMPLOYEE database

// Define field names which were not defined in the input scheme

DEF_F(FULL_N);
DEF_F(EID);

int main(int argc, char* argv[]) {
    const char* s = (EMPLOYEE/(DEPTNUM>3 && SALARY<=100000))  // Select a tuple subset
    [FIRST_N, LAST_N,
    FULL_N(cat(LAST_N," ", FIRST_N)),
    EID(ID)
    ].asSQL();
    // ... execute the SQL query in s using, e.g., ADO.
    return 0;
}
```

FIRST_N, LAST_N, FULL_N (computed from FIRST_N and LAST_N), and EID (which is just field ID renamed).

Note that the ARA expression cat(LAST_N," ", FIRST_N) produces a new (anonymous) field whose content is computed by concatenating three strings. The function call operator is then used to associate field name FULL_N with the result of this computation. Similarly, expression EID(ID) uses this operator for field renaming.

After this query object is created, its function member asSQL() is invoked (in line 16) to convert it into an equivalent SQL statement ready for execution:

```sql
select
    FIRST_N, LAST_N,
    concat(LAST_N," ", FIRST_N) as FULL_N,
    ID as EID
from EMPLOYEE where DEPTNUM > 3 and SALARY <= 100000;
```

This statement is assigned, as a string, to variable `s`.

As we saw, the usual C++ operators including comparisons and logical operators may be used in selection condition and in making the new fields. Table 2.2 summarizes the ARA equivalents of the main operators of relational algebra.
As can be seen in the table, the operators of relational algebra can be written in C++, using either a global function, a member function, or (if the user so chooses) with an intrinsic C++ (overloaded) operator: selection in relational algebra is represented by \( \text{operator } / \), projection by \( \text{operator } [ ] \), union by \( \text{operator } + \), difference by \( \text{operator } - \), natural join by \( \text{operator } * \), left join by \( \text{operator } <\), right join by \( \text{operator } >\), and renaming by the function call \( \text{operator } () \).

ARA does not directly support Cartesian product. Since the join of two relations with no common fields is their cross product, this operation can be emulated (if necessary) by appropriate field renaming followed by a join.

The translation of any relational algebra expression into C++ is quite straightforward. Figure 2.6.2 shows how a query object for finding the managers of departments of divisions in the city of Haifa can be generated using overloaded operators, global functions and member functions.

The composition of query objects with RAT is type safe—an attempt to gen-

---

<table>
<thead>
<tr>
<th>Relational Algebra Operator</th>
<th>ARA Operator</th>
<th>ARA Function</th>
<th>SQL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection ( \sigma_c R )</td>
<td>( R/c )</td>
<td>( \text{select}(R, c) ) ( R.\text{select}(c) )</td>
<td>( \text{select } * ) ( \text{from } R ) ( \text{where } c )</td>
</tr>
<tr>
<td>projection ( \pi_{f_1,f_2} R )</td>
<td>( R[f_1,f_2] )</td>
<td>( \text{project}(R, (f_1,f_2)) ) ( R.\text{project}((f_1,f_2)) )</td>
<td>( \text{select } f_1, f_2 ) ( \text{from } R )</td>
</tr>
<tr>
<td>union ( R_1 \cup R_2 )</td>
<td>( R_1 + R_2 )</td>
<td>( \text{union}(R_1, R_2) ) ( R_1.\text{union}(R_2) )</td>
<td>( R_1 ) ( \text{union } R_2 )</td>
</tr>
<tr>
<td>difference ( R_1 \setminus R_2 )</td>
<td>( R_1 - R_2 )</td>
<td>( \text{subtract}(R_1, R_2) ) ( R_1.\text{subtract}(R_2) )</td>
<td>( R_1 - R_2 )</td>
</tr>
<tr>
<td>(natural) join ( R_1 \bowtie R_2 )</td>
<td>( R_1 \bowtie R_2 )</td>
<td>( \text{join}(R_1, R_2) ) ( R_1.\text{join}(R_2) )</td>
<td>( R_1 ) ( \text{join } R_2 )</td>
</tr>
<tr>
<td>left join ( R_1 \bowland R_2 )</td>
<td>( R_1 \bowland R_2 )</td>
<td>( \text{left_join}(R_1, R_2) ) ( R_1.\text{left_join}(R_2) )</td>
<td>( R_1 ) ( \text{left join } R_2 )</td>
</tr>
<tr>
<td>right join ( R_1 \broand R_2 )</td>
<td>( R_1 \broand R_2 )</td>
<td>( \text{right_join}(R_1, R_2) ) ( R_1.\text{right_join}(R_2) )</td>
<td>( R_1 ) ( \text{right join } R_2 )</td>
</tr>
<tr>
<td>rename ( \rho_{a/b} R )</td>
<td>( b(a) )</td>
<td>( \text{rename}(a, b) ) ( a.\text{rename}(b) )</td>
<td>( a ) ( \text{as } b )</td>
</tr>
</tbody>
</table>
Fig. 2.6.2 Three alternative C++ expressions to compute a query object that, when evaluated, finds the managers of department in divisions located in the city of Haifa: using (a) overloaded operators (b) global functions, and (c) member functions.

```c++

(a) Operator overloading version

project(
    select(
        join(DIVISION, DEPARTMENT),
        (CITY == "Haifa")
    ), MANAGER)

(b) global functions version

DIVISION
    .join(DEPARTMENT)
    .select(CITY == "Haifa")
    .project(MANAGER)

(c) member functions version
```

Erate illegal queries will result in a compile-time error. Thus, expressions $q_1 + q_2$ and $q_1 - q_2$ will fail to compile unless $q_1$ and $q_2$ are query objects with the same set of fields. Similarly, it is illegal to project onto fields that do not exist in the relation, or select upon conditions that include such fields.

### 2.6.2 Storing Query Objects in Variables

A single C++ statement was used in Figure 2.6.1 to generate the desired query object. But, as can be seen in this example, as well as in Figure 2.6.2, a single C++ expression for generating a complex query objects might be a bit cumbersome. Moreover, there are cases in which a query object must be created incrementally.

For the purpose of creating a query object in several steps, RAT makes it possible to record intermediate objects in variables. In this recording, it is important to remember the significance of types: A query object can be assigned to a variable only if the type of this variable represents the same type of results as the query object.
Figure 2.6.3 makes use of query variables assignment to recode our motivating example, Figure 2.4.1, in RAT. Line 2 defines variable e which is initialized to the

\[ \text{Fig. 2.6.3 A rewrite of function get_employees using AraRat.} \]

```cpp
const char* get_employees(short dept, char* first) {
    DEF_V(e,EMPLOYEE);
    if (first != null)
        e /= (FIRST_N == first);
    if (dept > 0)
        e /= (DEPTNUM == dept);
    return e[FIRST_N,LAST_N].asSQL();
}
```

EMPLOYEE primitive query object. This line uses the macro \( \text{DEF_V} \), defined as

```cpp
#define DEF_V(var_name, exp) typeof(exp) var_name = exp}
```

to record both type and content of EMPLOYEE into variable e. Note that the second parameter to the macro is used twice: first for setting the type of the new variable by means of Gnu C++ \([39]\) operator typeof, and again, to initialize this variable. Hence, variable \( e \) can represent queries that return a relation with the same fields as EMPLOYEE. Initially, the evaluation procedure stored in this variable is an abstract encoding of the instruction to take all tuples of EMPLOYEE, but lines 3 through 6 modify it to specify a refined selection condition as dictated by get_employees’s parameters.

To complete the generation of the query object, we should evaluate the expression \( e[FIRST_N,LAST_N] \) representing the step of eliminating all fields but FIRST_N and LAST_N. Obviously, the type of this expression is not the same as the type of \( e \). The expression on Line 7 creates a temporary value whose type reflects the structure of the query result.

The example uses the abbreviated assignment operator (\texttt{operator /=}) to modify the query variable \( e \). This operator, that may be thought of as an extension of relational algebra, is legal, since selection does not change the type of the result. Similarly, RAT offers abbreviated assignment operators for union and subtraction, so that expressions such as \( e1+=e2 \) or \( e2-=e1 \) are legal whenever the type of result of \( e1 \) and \( e2 \) is the same.

The example uses method \texttt{asSQL()} of the query object to get an SQL statement representing the query. Another piece of information that can be obtained from a query object is the tuple type, a class that can store one tuple of the result.
relation. Given a query object \( e \), the following defines an array of pointers to object of \( e \) tuple type class, ready for storing the result of the query.

\[
\text{TUPLE}_T(e) \rightarrow \text{result} = \text{new} \ \text{TUPLE}_T(e)\times[\text{db\.result\_size()}];
\]

### 2.6.3 An Elaborate Example

Finally, we show a simple program to compute the names of employees who earn more than their managers. (This example was used, e.g., in the work on Safe Query Objects [27].) The code could have been written as a single expression of relational algebra. Instead, Figure 2.6.4 shows the computation in three stages.

![Fig. 2.6.4 Using ARA\_RAT to find the names of employees who earn more than their managers.](image)

```c
1 DEF_F(M\_SALARY);
2 const char* anomalous() {
3  DEF_V(e, (EMPLOYEE * DEPARTMENT[DEPTNUM(ID),MANAGER]));
4  DEF_V(m, EMPLOYEE[MANAGER(ID),M\_SALARY(SALARY)));
5  return ((e*m)[FIRST\_N,LAST\_N]/(SALARY>M\_SALARY)).asSQL();
6 }
```

Line 4 demonstrates how renaming is used to prepare the field names in a relation for a join operation: field ID in DEPARTMENT is renamed to DEPTNUM so that the subsequent join operation will yield the manager number of each employee. (Note that

```c
EMPLOYEE * DEPARTMENT[DEPTNUM(ID),MANAGER]
```

demonstrates how the common SQL phrase “\( T_1 \) join \( T_2 \) on \( T_1.f_1 = T_2.f_2 \)” is written using ARA\_RAT. A three-argument function joinOn that does the same, but without explicitly naming all fields, is a feasible addition to ARA\_RAT.)

Then, in line 5 we rename the fields in relation EMPLOYEE so that they can be used as descriptives of managers. Finally, line 6 does another join, to determine manager’s salary, projection of the desired fields, and then a selection of employees who earn more than their managers.

Observe that many non-useful fields are live in intermediate steps of the computation of Figure 2.6.4. For example, after line 4, the query variable \( e \) contains all EMPLOYEE fields, some of which, as LOCATION, are not used in the next steps of the query construction. No efficiency concern is to be made here: the computation
is of a query, rather than its result. Only the final result is submitted to optimized execution by the database engine.

2.7 The Ara Little Language

Having seen a number of examples, it is time for a more systematic description. Instead of a software manual, we use a language metaphor to explain how the Ararat system works: the user is expected to think of the system as an extension of the C++ programming language with a new little language, Ara, designed for composing SQL queries. Ara augments the mathematical formalism of relational algebra with features designed to enhance usability, and for integration with the host programming language. This section first describes the grammar and then highlights some of the semantics of this little language. The next section describes the techniques we used for realizing the grammar and the semantics within the framework of C++ templates, and hints how these techniques might be used for realizing other little languages.

2.7.1 Syntax

Ara extends C++ to allow a programmer to include any number of Ara definitions and statements within the code. The BNF grammar of definitions is given in Figure 2.7.1.

Fig. 2.7.1 The grammar of Ara. Part I: Schema Definition.

Definition ::= DEF_F(Field) | DEF_V(Var, Exp) | DEF_R(Relation, (Schema))
Schema ::= Field/Type [, Schema]
Type ::= INT | SMALLINT | BOOL | STRING | ...

The first line of the figure indicates that there are three kinds of definitions:

1. Field definitions are used to define the field label space of relational algebra. Each field name must be defined precisely once. For example, the following defines the two fields names used in the last relation of our running example (Table 2.1).
DEF_F(DIVNUM); DEF_F(CITY);

Field names are untyped until bound to a relation.

2. **Relation definitions**, which are similar to SQL’s *Data Definition Language* (DDL) statements, are used for specifying a database schema, by declaring relation names and the list of field-type pairs included in each. For example, the schema of the last relation of our running example, is specified by `DEF_R(DIVISION, (DIVNUM/SMALLINT, CITY/STRING))` (Such definitions are normally generated by the DB2ARA tool but evidently can be easily produced by hand.)

3. **Variable definitions**, made with the help of a `DEF_V` call, create a new C++ variable initialized to a relational algebra expression `Exp`. The full syntax of such expressions is given below in Figure 2.7.2, but any relation defined by `DEF_R` is also an expression. The statement `DEF_V(e, EMPLOYEE)` in Figure 2.6.3 thus defines `e` to the (trivial) relational algebra expression `EMPLOYEE`. Variable definition encodes in the type of variable the set of accessible fields in the expression.

Figure 2.7.2 gives a (partial) syntax of ARA statements.

---

**Fig. 2.7.2** The grammar of ARA. Part II: statements.

```
Statement ::= Exp; | Var+=Exp; | Var=Exp; | Var/=Cond;
     | Exp«Exp | Exp»Exp | Exp/Cond | Exp [Vocabulary]
Cond ::= Scalar
Scalar ::= C++ variable | C++ literal | Field
     | Scalar & Scalar | Scalar | ! Scalar
     | Scalar + Scalar | Scalar * Scalar | ~ Scalar
     | Scalar > Scalar | sin(Scalar) | cat(Scalar, Scalar)
Vocabulary ::= FieldOptInit [, Vocabulary]
FieldOptInit ::= Field | Field (Scalar)
```

Terminals include `Relation`, `Field` and `Var` which are C++ identifiers, respectively representing a name of a relation in the database, a field name, and a variable storing a query object.

The most important non-terminal in the grammar is `Exp`, which denotes a relational algebra expression obtained by applying relational algebra operators to
atomic relations. An \texttt{Exp} can be used from C++ in two main ways: first, such an expression responds to an \texttt{asSQL()} method; second, any such expression can be passed to the \texttt{TUPLE_T} macro, which returns a \texttt{record}, a class that has no methods and that has all of its fields as \texttt{public} [46], containing all accessible fields in this relation.

An \texttt{Exp} may involve (i) relations of the database, (ii) C++ variables storing other \texttt{Exp}s, or (iii) field names. All three must be previously defined.

An \texttt{ARA} statement can be used anywhere a C++ statement is legal. It can be an \texttt{Exp}, or it may modify a C++ variable (defined earlier by a \texttt{DEF_V}) by applying to it the union or subtraction operators of relational algebra. Similarly, a statement may apply a selection operator to a variable based on a condition \texttt{Cond}. It is not possible to use the join and projection operators to change a variable in this fashion, since these operators change the list of accessible fields, and hence require also a change to the variable type.

An \texttt{Exp} is composed by applying relational algebra operators, union, subtraction, selection, projection and the three varieties of join to atomic expressions. Atomic expressions are either a C++ variable defined by \texttt{DEF_V} or a relation defined by \texttt{DEF_R}. An \texttt{Exp} may appear anywhere a C++ expression may appear, but it is used typically as receiver of an \texttt{asSQL()} message, which translates the expression to SQL.

\texttt{Cond} is a \texttt{Scalar} expression which evaluates to an SQL truth value. The type system of \texttt{ARA} is similar to that of SQL, i.e., a host of primitive scalar types, including booleans, strings, integers, and no compound types. Scalar expressions of \texttt{ARA} must take one of these types. They are composed from C++ literals, C++ variables (which must be of one of the C++ primitive types or a "\texttt{char *}"), or \texttt{RAT} fields. Many logical, arithmetical and builtin functions can be used to build scalar expressions. Only a handful of these are presented in Figure 2.7.2.

Finally, note that the projection operation (operator \texttt{[ ]}) involves a \texttt{Vocabulary}, which is more general than a simple list of field names. As in SQL, \texttt{ARA} allows the programmer to define and compute new field names in the course of a projection. Accordingly, a \texttt{Vocabulary} is a list of both uninitialized and \texttt{initialized} fields. An uninitialized field is simply a field name while an initialized field is a renamed field or more generally, a field initialized by a scalar expression.
2.7.2 Semantics

RAT defines numerous semantic checks on the ARA little language. Failure in these triggers an appropriate C++ compilation error. In particular, RAT applies symbol table lookups and type checking on every scalar expression.

For example, in a selection \( e/c \) expression, RAT makes sure that every field name used in the scalar expression \( c \) exists in the symbol table of \( e \); RAT then fetches the type of these fields, and applies full type checking of \( c \), i.e., that the type signature of each operator matches the type of its operands; finally, if \( c \)’s type is not boolean, then the selection is invalid.

Other checks are shown in Figure 2.7.3.

**Fig. 2.7.3** Some semantic checks applied by RAT.

1. In union \( e_1+e_2 \), and in subtraction \( e_1-e_2 \), the sets of fields of \( e_1 \) and \( e_2 \) must be the same.

2. In \( e_1+e_2, e_1-e_2, e_1*e_2, e_1<<e_2 \) and \( e_1>>e_2 \), if a field is defined in \( e_1 \) with type \( \tau \), then either this field is not defined in \( e_2 \), or it is defined there with the same type \( \tau \).

3. In a selection, \( e/c \), expression \( c \) is a properly valid expression of boolean type.

4. There exists an evaluation sequence for a vocabulary, i.e., the initializing expression of any initialized field does not depend, directly or indirectly on the field itself.

5. In using a Vocabulary in a projection it is required that
   (a) all initialized fields are *not* found in the symbol table of the projected relation;
   (b) all uninitialized fields exist in this symbol table;
   (c) If an initializing expression uses a field that does not occur in the vocabulary, then this field exists in the projected relation.
### Tab. 2.3 Realizing compiler stages with template libraries.

<table>
<thead>
<tr>
<th>Compiler Stage</th>
<th>Expression Templates [129] / SEMT [45]</th>
<th>RAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Analysis</td>
<td>C++ Compiler</td>
<td>C++ Compiler</td>
</tr>
<tr>
<td>Parsing</td>
<td>C++ Compiler</td>
<td>Template Engine</td>
</tr>
<tr>
<td>Type Checking</td>
<td>Single-sorted</td>
<td>Template Engine</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Template Engine</td>
<td>Program runtime</td>
</tr>
</tbody>
</table>

### 2.8 System Architecture

The ARA little language is implemented solely with the C++ template mechanism, and without modifications to the compiler of the host language, nor with additional pre- or post-processing stages. This section explains how this is done. In Section 2.8.1, we explain the general technique of representing the executional aspects of an expression of relational algebra as a C++ runtime value, without compromising type safety. Section 2.8.2 then discusses some of the main components of the RAT architecture. Section 2.8.3 demonstrates the technique, showing how these components cooperate to achieve compile-time assurance that only boolean expressions are used for selection. Finally, Section 2.8.4 discusses the two compiler extensions that RAT uses.

#### 2.8.1 Combining Compile-time and Runtime Representations

As explained above, query objects do not use the standard representation of symbolic expressions as types (described in Section 2.3.3) for representing the expressions of relational algebra. This technique is however used for the representation of boolean and arithmetical expressions used in selection and in defining new fields.

Table 2.3 compares the compiler architecture (so to speak) of RAT with that of the expression templates library and SEMT. (Table rows represent compilers’ main stages.)

It is an inherent property of template-based language implementation that the lexical analysis is carried out by host compiler. Similarly, since no changes to the host compiler are allowed, the syntax of the little language is essentially also that of C++, although both expression templates and ARA make extensive use of operator overloading to give a different semantics to the host syntax to match the application needs.

The main objective of expression templates and SEMT is runtime efficiency.
Accordingly, the type system in these is single-sorted, and code is generated at compile-time by the template engine. Conversely, ARA is designed to maximize compile-time safety, and includes its own type- and semantic-rules, as was described in Section 2.7.2. ARA has a multi-sorted type system, and non-trivial semantic rules, which are all applied at the time the host C++ language is compiled. This design leads to the delay of code generation (production of the desired SQL statement) to runtime. An advantage of this delay is that the same code fragment may generate many different statements, depending on runtime circumstances, e.g., it may use C++ parameters. To make this possible, the structure of a scalar expression is stored as a runtime value. Types (which can be thought of as compile-time values) are used for recording auxiliary essential information, such as the list of symbols which this expression uses. These symbols are bound later (but still at compile-time) to the data type dictionary of the input schema.

This technique (together with the delayed execution semantics of query objects) makes it possible to use the same field name, possibly with distinct types, in different tables. But, the main reason we chose this strategy is to allow assignments as \( e /= (\text{DEPTNUM} == \text{dept}); \) (line 6 of Figure 2.6.3), which would have been impossible if the type of \( e \) reflected the Abstract Syntax Tree of the query rather than the result type of the query.

Curiously, this technique supports what may seem paradoxical at first sight: a selection based on fields that were projected out, making it possible to rewrite Figure 2.6.3 as Figure 2.8.1.

![Fig. 2.8.1](image)

A re-implementation of function `get_employees` (Figure 2.6.3) in which selection is applied after projection.

```c
1 const char* get_employees(short dept, char* first) {
2   DEF_V(e,EMPLOYEE[FIRST_N,LAST_N]);
3   if (first != null) e /= (FIRST_N == first);
4   if (dept > 0) e /= (DEPTNUM == dept);
5   return e.asSQL();
6 }
```

Observe that in line 4 we apply a selection criterion which depends on field `DEPTNUM`, which was projected out in line 2. This is possible since the type of each query entity encodes two lists:

1. **Active Fields.** this is a list with names and types of all fields in the tuples computed by evaluating the query; and
2. **Symbol Table.** This includes the list of all fields against which a selection criterion may be applied. In particular, this list includes, in addition to the active fields, fields which were projected out.

**Comment.** This crisp distinction between runtime and compile-time representation does not apply (in our current implementation of RAT) to scalar expressions. Consequently, it is not possible to define a C++ variable storing a selection condition, and then modify this expression at runtime. Each boolean expression in ARA has its own type.

### 2.8.2 Concepts in RAT

A *type concept* [7] (or for short just a concept) is defined by a set of requirements on a C++ type. We say that a type *models* a concept, if the type satisfies this concept’s requirements. Thus, a concept defines a subset of the universe of all possible C++ types. The notation \( C_1 \preceq C_2 \) (concept \( C_1 \) refines concept \( C_2 \)) means that every type that models \( C_1 \) also models \( C_2 \).

Concepts are useful in the description of the set of types that are legible parameters to a template, or may be returned by it. However, since the C++ template mechanism is untyped (in the sense that the suitability of a template to a parameter type is checked at application time), concepts are primarily a documentation aid. (Still, there are advanced techniques [87,111,135] for realizing concepts in the language in such a way that they are checked at compile-time.) A language extension to support concepts [54] is a candidate for inclusion in the upcoming revision of the ISO C++ standard.

The RAT architecture uses a variety of concepts for representing the different components of a query, including field names, field lists, conditions etc. Table 2.4 summarizes the main such concepts. Comparing this table with the language grammar (Figures 2.7.1 and 2.7.2), we see that concepts (roughly) correspond to non-terminals of the grammar.

The most fundamental concept is \( F \), which represents symbolic field names. The vocabulary concept \( V \) represents a set of fields (such sets are useful i.e., for relational algebra projection). The last cell in the first row of the table states that a C++ expression of the form \( v_1, v_2 \) (applying `operator` to values \( v_1 \) and \( v_2 \)) where \( v_1 \) and \( v_2 \) belong in \( F \), returns a value in \( V \). The type of this returned value represents the set of types of \( v_1 \) and \( v_2 \). For example, the expression \( \text{FIRST}_N, \text{LAST}_N \) belongs in \( V \), and it records the set of these two symbols.
**Tab. 2.4** The main concepts in the RAT architecture.

<table>
<thead>
<tr>
<th>Concept Name</th>
<th>Purpose</th>
<th>Sample Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$ Field</td>
<td>Symbolic field name.</td>
<td>$F, F : V$</td>
</tr>
<tr>
<td>$I$ Initialization</td>
<td>A symbolic field name, along with an initialization expression, $F \preceq I$.</td>
<td>$I, I : V$ $F(S) : I$</td>
</tr>
<tr>
<td>$V$ Vocabulary</td>
<td>A set of (possibly initialized) symbolic field names.</td>
<td>$V, I : V$ $I, V : V$</td>
</tr>
<tr>
<td>$S$ Scalar</td>
<td>An expression evaluating to a scalar, e.g., string, boolean, integer, obtained by applying arithmetical, comparison and SQL-like functions to fields, literals and variables, $F \preceq S$.</td>
<td>$S+S : S$ $S*S : S$ $\text{cat}(S, S) : S$ $S\geq S : S$ $S \mid</td>
</tr>
<tr>
<td>$R$ Relation</td>
<td>An expression in enriched relational algebra evaluating to a relation.</td>
<td>$R*R : R$ $R+R : R$ $R-R : R$ $R[V] : R$ $R/S : R$</td>
</tr>
</tbody>
</table>

The notation $A \diamond B : C$ where $A$, $B$ and $C$ are concepts and $\diamond$ is a C++ operator means that the library defines a function template overloading the operator $\diamond$, such that the application of $\diamond$ to values in kinds $A$ and $B$ returns a value of concept $C$. This notation is naturally extended to unary operators and to the function call operator.

Types that model $F$ are singletons. In our running example, the value \texttt{FIRST\_N} is the unique instance of the type representing the symbol \texttt{FIRST\_N}. The macro invocation \texttt{DEF\_F(FULL\_N)} (line 6 Figure 2.6.1) defines a new type that models $F$ along with its single value. Henceforth, for brevity’s sake we shall sacrifice accuracy in saying that a value belongs in a certain concept meaning that this value’s type models this concept. This convention will prove particulary useful when talking about singleton types. Thus, we say that this macro invocation defines the value \texttt{FULL\_N} in concept $F$.

The concept $I$ represents initialized fields, necessary for representing expressions such as

\[
\texttt{FULL\_N(cat(LAST\_N, ",", \ FIRST\_N))} \tag{2.1}
\]
A type modeling $I$ has two components: a field name and an abstract representation of the initializing expression.

Concept $S$ represents scalar expressions used in an initialization expression and in selection, e.g., $\text{cat}(\text{LAST}_N, \text{"", } \text{FIRST}_N)$ is in $S$. In writing $F(S) : I$ in the table we indicate that expression (2.1) which applies the overloaded function call operator of field $\text{FULL}_N$ to $\text{cat}(\text{LAST}_N, \text{"", } \text{FIRST}_N)$ is in $I$.

Since $F \subseteq S$ we have that the function call operator can be used in particular to do relational algebra-renaming, writing, e.g., $\text{EID}(\text{ID})$ (line 14 in this figure).

Another instance of $S$ is $(\text{DEPTNUM} > 3 \&\& \text{SALARY} \leq 100000)$, an expression shown in line 10, which demonstrates that scalar expressions may involve literals.

Concept $R$ is used for representing relational algebra expressions. The rightmost cell in the last row of the table specifies the semantics of union, subtraction, cross product, selection, and projection. We can see that RAT enriches the semantics of relational algebra. For example, vocabularies may include initialized fields. The initialization sequence of such fields specifies the process by which this field is computed from other fields during projection.

### 2.8.3 Type Safety of Selection and Projection Expressions

Now that the main concepts of the RAT architecture have been enumerated, we turn to describing how these concepts are used at both compile-time and runtime to realize the integration of relational algebra into C++. The description is as high-level as possible, although some of the technical details do pop out.

#### Managing Scalar Expressions

At runtime, a scalar expression is represented as a value of type `S_TREE`. Type `S_TREE` is the root of a type hierarchy that does a standard textbook implementation of an expression tree, with classes such as `S_BINARY` (for binary operators), and `S_UNARY` (for unary operators), to represent internal nodes. Leaves of the tree belong to one of two classes: (i) `S_LIT`, which store the value of C++ values participating in the expression, and (ii) `S_FIELD` representing a name of one of the fields in the underlying relational algebra.

The representation is untyped in the sense that the actual evaluation of each node in the tree may return any of the supported types of scalar expressions, including booleans, strings and integers. In contrast to the standard representation
of an expression tree, the nodes of this tree do not have an evaluation function. Instead, the class S_TREE has a pure virtual function char *asSQL(). This function is implemented in the inheriting classes, to return the SQL representation of the expression by a recursive traversal of the subtree.

Thus, the evaluation of an S_TREE is carried out by translating it to SQL. This translation is always performed in the context of translating a relational algebra expression, which contains the scalar expression. At the time of the translation, S_LIT leaves are printed as SQL literals, while S_FIELD leaves are printed as SQL field names.

Figure 2.8.2 shows what a class $S$ modeling concept $S$ looks like.

![Figure 2.8.2](image)

Fig. 2.8.2 The main component of a type modeling concept $S$.

```cpp
class S {
  public:
    const S_TREE *t; // Expression tree of $S$
    typedef ... TYPS; // Compile-time representation of $t$
    typedef ... FLDS; // List of fields used in $t$
};
```

As seen in the figure, each such type has a data member named $t$ which stores the actual expression to be evaluated at runtime.

For the purpose of compile-time type checking of the scalar expression, when using it for projection or selection, each such type has two compile-time properties, realized by a `typedef`:

1. property `TYPS` is a compile-time representation of the content of $t$, i.e., `TYPS` is tree-structured type, with the same topology as $t$. However, instead of literal values and addresses of variables, which are difficult to represent in the type system, this compile-time representation stores just their types.

2. property `FLDS` is a type which represents the list of field names that take part in this scalar expression. Each node in this list is a type modeling concept $F$.

A number of function templates (including many that overload the standard operators) are defined in RAT. All these functions generate types modeling $S$. Each concrete template function obtained by instantiating these templates computes the value of data member $t$ in the return value, and generates the `typedef`s `TYPS` and `FLDS` of the type of the result.
More specifically, if $S_1$ and $S_2$ are types that model $S$, then the return type of 
\begin{verbatim}
operator +(const &S1, const &S2)
\end{verbatim}
is a type $S$ modeling $S$ such that (i) the list $S::FLDS$ is the merge of $S1::FLDS$ and $S2::FLDS$, and (ii) the type tree $S::TYPS$ is rooted at a node representing an addition (i.e., a union in the relational algebra sense) of two type subtrees $S1::TYPS$ and $S2::TYPS$.

The actual value returned by a call \texttt{operator +(x, y)} (where the class of $x$ is $S_1$ and that of $y$ is $S_2$), is such that its $t$ field is of class $S\_PLUS$ and has two subtrees $x.t$ and $y.t$.

Note that types that model concept $S$ have compile-time properties that describe the expression. Therefore, these types cannot be reassigned with a different expression.

**Managing Relational Algebra Expressions**

As already mentioned, types that model concept $R$ have a runtime encoding of the procedure for evaluating the query and two main compile-time properties: an encoding of the schema of the result of a query, and a list of fields that can be used in a selection criterion.

When a selection operation on relation $r$ of type $R$ with scalar expression $s$ of type $S$ is encountered, $S$ is bound to $R$. Steps in this binding include:

1. A check that all fields in $S::FLDS$ are present in $R$.
2. Binding each field of $S$ to its type in $R$ to analyze the types of $S::TYPS$.
3. Issuing a compilation error, if there is a type mismatch between an operator and its operands or if the result type is non boolean.
4. Integration of the content of $s.t$ with $r$. This integration affects only the runtime value of $r$ and therefore reassigning the new value into the same variable $r$ is possible.

A projection operation is defined on relation $r$ of type $R$ and field list $v$ of type $V$ modeling concept $V$. There are two kinds of elements in $v$: uninitialized fields, each modeling concept $F$, and initialized fields, each modeling concept $I$. $RAT$ verifies that all the uninitialized fields are present in $R$ and all initialized fields are absent of $R$. $RAT$ also verifies that the initialized fields can be calculated when bound to $R$ using the same algorithm used in the selection operation. In addition,
the compile-time encoding of the result schema and the symbol table of \( R \) are updated, which means that the result of a projection operation is an object of a new type which cannot be assigned to \( R \).

### 2.8.4 Compiler Extensions

Portability and simplicity were primary among our design objectives: ARARAT minimizes the use of external tools, and can be used (at its core) with any standard [114] C++ compliant compiler. The DB2ARA pre-processor translates a database schema into ARA declarations. This process, which can be thought of as writing type definitions in a header file, is not difficult and can be carried out by hand.

For more advanced features, we rely on two small extensions: the `typeof` operator and the `__COUNTER__` macro.

#### Extracting a variable’s type.

An incremental generation of query object requires that intermediate objects are stored in variables. It is necessary to record both the type and the content of these objects. ARARAT extracts the type of a query object with the non-standard `typeof` pseudo operator. Thus, macro `DEF_V` in Section 2.6 creates a new variable `var_name`, which has the type of `exp`, and initializes this variable with the content of `exp`.

The `typeof` operator is a pseudo operator, since instead of returning a value, it returns the type of the argument, which can be used anywhere a type is used. Like `sizeof`, this operator is evaluated at compile-time. This operator is found in, e.g., all Gnu implementations [39] of the language; its significance was also recognized by the C++ committee, which is considering a similar mechanism for querying the type of an expression, namely the `decltype` operator, as a standard language extension.\(^6\)

An alternative implementation of the `DEF_V` macro could rely on another extension that is being considered for the next revision of C++—the `auto` keyword for indicating that the compiler should deduce the type of a variable from its initializer expression. Using this extension, we could rewrite the definition of macro `DEF_V` as

\[
\texttt{#define \_DEF\_V(var\_name, exp) \_auto\_ var\_name = exp}
\]

and this definition would have been equivalent to the one described in Section 2.6.

\(^6\)www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/
With neither `typeof` nor `auto`, ARARAT can still produce query objects, but in order to store these in variables the user must manually define appropriate types. The compiler still checks that these type definitions are correct and consistent with the query objects.

Field ordering.

As shall be explained in Section 2.9, the `TUPLE_T` macro generates a record whose field names and types correspond to the fields in the result relation of a query object by a recursive template instantiation. In order to generate equivalent types for equivalent relations, it is essential that this instantiation chain be applied in a predetermined order. To this end, a total ordering relation is placed on fields. In our implementation, this order is realized by a unique integral value associated with every field name. The identifier is generated using another C++ application extension, the `__COUNTER__` macro. This macro is a compile-time counter, evaluating to an integer and is incremented on each use. It is supported by Microsoft compiler [103], and is scheduled to be included in version 4.3 of g++ (pending approval on the GCC steering committee [123]). Using this macro ensures that each field has a constant identifier and thus every relation has a unique representation.

Without the `__COUNTER__` macro, ARARAT must resort to using the standard `__LINE__` macro as an alternative means for imposing a total order on fields. The limitation placed on the user is that all fields must be defined on different lines in the same source file.

2.9 Structural Typing in ARARAT

2.9.1 Structural Equivalence

Relational algebra puts no significance to fields’ order: Two schemas are equivalent if they consist of the same set of fields, regardless of the order in which these fields were defined or represented internally. The `TUPLE_T` macro, which generates a record whose field names and types correspond to the fields of the result relation of a given query object, should reflect this equivalence relation. Consider again the ARARAT statements

```c
TUPLE_T(DIVISION * DEPARTMENT) x;
TUPLE_T(DEPARTMENT * DIVISION) y;
```
which define variables $x$ and $y$ to be of a record whose field names and types are the union of the field sets of DIVISION and DEPARTMENT. Macro TUPLE_T must make variables $x$ and $y$ assignment compatible, despite the nominal nature of definition of record types, that is struct and class, in C++. It is not sufficient that the type of variable $x$ is a record with the correct set of fields and their types; it is required that unlike what is shown in Figure 2.3.2, this record definition should obey structural typing rules.

Recall that each query object has a compile-time representation of the set of fields. The template based macro TUPLE_T iterates over that set of its parameter, generating an inheritance chain, with one class in this chain for each field. So, if the fields set contains three fields, an ID of type int, a NAME of type string, and a ZIP code of type int, then the resulting inheritance chain produced by RAT has four struct types, whose outline is as in Figure 2.9.1.

![Fig. 2.9.1 An inheritance chain composing a record of fields.](image)

```c++
struct I0 {
    // empty
};

struct I1: public I0 {
    int ID;
};

struct I2: public I1 {
    string NAME;
};

struct I3: public I2 {
    int ZIP;
};
```

We see in the figure, that type I0 is the basis of the chain, containing no fields, while types I1, I2 and I3 add fields ID, NAME and ZIP in order. The type I3 at the end of the inheritance chain is a record with all the required fields.

The actual invocation of the TUPLE_T macro indeed generates types similar to I0, I1, I2, and I3 as outlined in Figure 2.9.1, except that these types do not carry names. Instead, the macro makes use of a class template type which inherits from one of its arguments, to generate each of the types in the inheritance chain. Each of the types in this inheritance chain is an anonymous instantiation of this
template. The total order imposed on field types is used to ensure that the fields are sorted, and that the inheritance chain is always created in the same order. This, together with the observation that template instantiation is structural, guarantees the correct working of `TUPLE_T`, and that variables \( x \) and \( y \) are of the same type.

### 2.9.2 Structural Subtyping

A natural question to ask at this point is whether one can use template instantiation to emulate not only structural equivalence, but also structural subtyping. Say that the set of fields in a query object \( q_1 \) is contained in the set of fields of query object \( q_2 \), and that two types \( T_1 \) and \( T_2 \) are defined by

```cpp
typedef TUPLE_T(q1) T1;
typedef TUPLE_T(q2) T2;
```

Then, the question is whether type \( T_1 \) can be defined in such a manner that values of type \( T_2 \) can be assigned to it, while omitting the spurious fields? A more demanding desire is that type \( T_1 \) is a super type of \( T_2 \). The technique used for ensuring structural equivalence does not achieve any of these ends is as follows: An inheritance chain containing fields \( ID \) and \( NAME \) in that order will generate a super type of the type \( I_3 \) as defined in Figure 2.9.1, but, an inheritance chain for (say) the field \( ID \) and \( ZIP \) will not be a super type of \( I_3 \).

As it turns out, it is possible to generalize the inheritance chain idea to what may be called an “inheritance lattice”, in such a way that subtyping is preserved. The way this may be achieved is demonstrated in Figure 2.9.2.

#### Fig. 2.9.2 An inheritance lattice composing a record of every subset of fields.

```cpp
struct TOP { }

struct ID: public virtual TOP { int ID; }
struct NAME: public virtual TOP { string name; }
struct ZIP: public virtual TOP { int ZIP; }

struct ID_ZIP: public virtual ID, public virtual ZIP {};
struct ID_NAME: public virtual ID, public virtual NAME {};
struct ZIP_NAME: public virtual ZIP, public virtual NAME {};

struct X: public virtual ID_ZIP, public virtual ID_NAME {};

struct ID_ZIP_NAME:
    public virtual X, public virtual ZIP_NAME {};
```

48
In the figure, we see a struct for every subset of the three fields: ID, NAME and ZIP. For example, the type ID_NAME is a record with two fields ID and NAME. Moreover, every such record inherits, directly or indirectly, from each record containing a subset of its fields. Thus, ID_ZIP_NAME inherits from all records with two fields, ID_ZIP, ID_NAME and ZIP_NAME, from all records of one field, ID, NAME, ZIP and from the TOP, the record with no fields.

The generation of record types can be done by the usual techniques of template programming, with, as before, templates that inherit from their arguments. A record with \( n \) fields should inherit from all \( n \) record types representing subsets with \( n - 1 \) fields. Indirect inheritance may be used to restrict inheritance fan out, and for using only templates with fixed number of arguments. In the figure, instead of having three immediate parents, type ID_ZIP_NAME inherits from two types, ZIP_NAME and an intermediate type \( X \) which in turn inherits from ID_ZIP and ID_NAME.

The solution demonstrated in Figure 2.9.2 is exponential. If a database uses \( n \) fields, then the number of principal types so generated is \( 2^n \). Auxiliary types, such as the type \( X \) in Figure 2.9.2 even add to this number. The impact on compilation time should be evident. Should this be a concern, one may resort to a method of approximating structural subtyping by using conversion operators instead of the explicit inheritance lattice.

This approach, pioneered by the work of Solodkyy, Jarvi and Mlaih [113], is based on overloading of the assignment operator, or of the type cast operator. This overload is done in such a way that an assignment in the correct direction is possible between every two types that stand in a structural subtyping relation. The clever observation made by Solodkyy, Jarvi and Mlaih is that one can use a conditional template expansion, (specifically the enable_if templates [69]) to test first whether two types stand in a subtyping relation, and only generate the conversion code if this is the case. To do this correctly, each record \( T_1 \) type defines a constructor template, which takes a type argument \( T_2 \), and a constructor argument of type \( T_2 \). Now, the participation of instantiation of this constructor template in the overloading resolution tournament is guarded by an enable_if template, as outlined by Figure 2.9.3.

Class VOID, defined in lines 2 through 4. in the figure is an auxiliary class, used by all record types. The POINTER typedef definition within that class is such that VOID::POINTER names the type void *.
Examine now the type $T_1$, defined in lines 7 through 20. In this type, there is a constructor template which given a type $T_2$, generates a constructor that takes an argument of type $T_2$ and creates a $T_1$ object out of it. The enable_if templates is then used to restrict the generation of this constructor. If the compile-time condition $\text{is_subtype}<T_2, T_1>$ evaluates to the boolean constant false, enable_if does not generate a valid type; in this case, the second argument to this constructor has no valid type, and the substitution-failure-is-not-an-error principle [128] prevents the constructor from being generated.

If however, $\text{is_subtype}<T_2, T_1>$ evaluates to true, then dummy, the second argument, receives the type VOID:POINTER, i.e., void *, the constructor template is then activated. Note that this second argument receives a default value of 0. Therefore, the constructor call $T_1(t_2)$ where $t_2$ is an instance of $T_2$ assigns this value to dummy, and compiles correctly. Moreover, if a C++ function expects an argument of type $T_1$, and the actual argument is of a type $T_2$ which happens to be a structural subtype of $T_1$, then the compiler will insert the appropriate constructor call. The said function will then appear to be polymorphically applicable to all structural subtypes of $T_1$. 

50
This technique does not carry the exponential price tag. But, it should be re-
membered that it is only an imitation of true subtype. It would fail, for example, if subtyping is examined through run time type information, or if a sequence of
conversions is required.

2.10 Discussion and Further Research

The ARARAT system demonstrates a seamless integration of the relational alge-
bra formalism with C++, which can be used to generate safe SQL queries (using
method asSQL()). Also, ARARAT makes it possible to generate a record type for
storing query results. The challenges in the implementation were in the restriction
on use of external tools, without introducing cumbersome or long syntax.

The compilation time of the program in Figure 2.4.1 is shorter than that of
Figure 2.8.1. The compilation of the program in Figure 2.4.1 took 1.04 seconds
while the program in Figure 2.8.1, which uses the Rat library that consists of
about 3000 lines of C++ code, compiled in 1.26 seconds. These time differences
do not constitute a main consideration in this sort of applications. But of course, a
comprehensive benchmark is required for comparing the runtime performance of
the two alternatives over a wide range of queries. The benchmark should measure
both the construction and execution time of queries. We venture to project and ex-
trapolate from the measurements we made that the finding will be that construc-
tion time is negligible, and that generating queries using template mechanisms does not
impose a significant performance penalty.

The current implementation support of dynamic queries is limited to modifi-
cations of a query object by applying selection, union and subtraction to it. It is
mundane to add support to dynamically created conditions, allowing to define a
selection condition with code such as shown in Figure 2.10.1.

Fig. 2.10.1 A possible ARARAT extension supporting dynamic conditions

```c
1   DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE);
2   if (dept > 0);
3       c &= (DEPTNUM == dept);
4   if (first != null)
5       c &= (FIRST_N == first)
6    ...
```

7On a 2.13GHz Intel(R), Pentium(R) M processor with 2 GB of RAM.
The statement `DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE);` is used in line 1 in the figure to encode the symbol table of `EMPLOYEE` into `c`. Then, in lines 2 through 5 the condition is refined.

Also easy in principle is the definition of “prepared SQL statements”, by storing memory addresses of C++ variables instead of actual content. The actual value of these variables is retrieved whenever the query is translated to SQL. Moreover, as hinted in brief in Section 2.8.4, RAT can easily generate function members for these queries which would allow dynamic changes of the parameters of a prepared statement.

Conversely, it should be stated that the extent of flexibility in dynamic queries is not as great as with programs not using ARARAT. For example, such programs may be able to generate a query that returns a user-specified subset of the fields of a given relation, whereas in ARARAT, every such subset must be available for at compile-time for inspection by the compiler.

In Section 2.8 we described non-trivial implementation techniques, by which both compile-time and runtime values are used for realization of the embedded language. These techniques, and the experience of managing symbol tables, type systems, and semantical checks with template programming, can be used in principle for introducing many other little languages to C++. Further research will probably examine the possibility of doing that, and in particular in the context of a little language for defining XML data. Presumably, the same AR representation can be used to generate not only SQL, but also directives for other database systems.

Another prime candidate for a little language to be added thus in C++ is the SQL language itself. Indeed, a user preferring explicit function names, as in Figure 2.6.2(b) and Figure 2.6.2(c) will find the resulting code similar to SQL. We contemplate extending this to support a more SQL-like syntax as done in, e.g., LINQ. Still, it should be remembered that, as evident by the LINQ experience, support of even the `select` statement can be only partial, mainly because the syntax is too foreign to that of the host language. This is the reason we believe that the advantage of building upon user’s familiarity with SQL is not as forceful as it may appear.

On the other hand, it should be clear how to extend ARARAT to support more features offered by the `select` statement, without imposing an SQL syntax. For example, we can easily extend the ARARAT syntax to support sorting and limits features.
of \texttt{select}, by adding, e.g., the following rules to Figure 2.7.2.

\[
\text{Exp ::= Exp.asort(Field) | Exp.dsort(Field) | Exp.limit(Integer)}
\]

The addition of support for \texttt{group by} clause is more of a challenge, since it requires a type system which allows non-scalar fields, i.e., fields containing relations.

Our work concentrated on selection and queries since these are the most common database operations. We demonstrated how these queries can be generated in a safe manner, and showed how RAT can define a receiver data type. The extension to support the generation of other statements in the Data Manipulation sub-Language (DML) of SQL does not pose the same challenges as those of implementing queries.

In a similar manner, this work does not deal with the many different variants of the SQL language. It is well known that commercial vendors introduce their own (some may say idiosyncratic) dialect in adoption of the SQL standard. Simple statements will be understood by all such implementations. However, realization of more advanced features, such as nested queries may be very different in different database implementation. The procedure of translating the content of a query object into an SQL statement must be cognizant of these differences. This demand can be compared to the requirement that an implementation of a high level programming language, specifically a compiler, must be aware of the target language. Writers of multi-targets compilers address this difficulty by producing object code in intermediate form, which is then translated to the specific machine code. The same idea applies to RAT: the internal form of a query object is an arbitrarily complex expression in relational algebra. The familiar post order traversal of such an expression yields an evaluation procedure whose elements are simple SQL queries, each of which operates on previously computed intermediate results, yielding another such result. The final such operation computes the entire query result. This process lends itself to a general purpose process of translation into an arbitrary SQL implementation. The cost is of course is in that less-sophisticated database implementation may miss optimization opportunities, when given such a sequence of operations. A specialized translator may be in place for such implementations.

We note that RAT does not take responsibility on the execution of statements, although it is possible to use it to define the data types which take part in the actual execution. We leave it to future research to integrate the \textit{execution} of queries
and other DML statements with C++. This research should strive for a smooth integration of the execution with STL. For example, an insert statement should be able to receive an STL container of the items to insert. Interesting, challenging and important in this context is the issue of integration of database error handling, transactions, and locking with the host language.

A fascinating direction is the application of ARA to C++ own data structures instead of external databases. Perhaps the best integration of databases with C++ is achieved by mapping STL data structures to persistent store.

Finally, note that work on integration of SQL with other languages can be also viewed as part of the generative programming line of research [9, 48, 70, 73, 105]. It is likely that other lessons of this subdiscipline can benefit the community in its struggle with the problem at hand. For example, it may be possible to employ the work on certifiable program generation [33] to prove the correctness of RAT.
Chapter 3

The Use of Overloading in JAVA Programs

Method overloading is a controversial language feature, especially in the context of Object Oriented languages, where its interaction with overriding may lead to confusing semantics. One of the main arguments against overloading is that it can be abused by assigning the same identity to conceptually different methods.

This chapter describes a study of the actual use of overloading in JAVA. To this end, we developed a taxonomy of classification of the use of overloading, and applied it to a large JAVA corpus comprising more than 100,000 user defined types.

We found that more than 14% of the methods in the corpus are overloaded. Using sampling and evaluation by human raters we found that about 60% of overloaded methods follow one of the “non ad hoc use of overloading patterns” and that additional 20% can be easily rewritten in this form. The most common pattern is the use of overloading as an emulation of default arguments, a mechanism which does not exist in JAVA.

Chapter Outline. The remainder of this chapter is organized as follows. Section 3.1 describes the overloading debate. Research questions, and an overview of the research are presented in Section 2.2. In Section 3.3 we describe the setting of our empirical evaluation, the results of the automatic analysis, the sampling and the employment of human evaluators. Section 3.4 presents the taxonomy of the kinds of overloading that may be found in actual code, as developed with the aid of the human experimenters. The reliability of the classification is studied in Section 3.6, which also lays the foundation for deduction of conclusions regarding
the entire corpus from the sample. The results of the classification according to this taxonomy are presented in Section 3.7. Section 3.8 concludes.

3.1 The Overloading Debate

208, 765, 973, 875, 851, the count of distinct admissible identifiers in early versions of C, may seem a fairly large number. Still, as large as this number is, it is infinitesimally small when compared to its JAVA counterpart. Yet, adequate identifier names are hard to come by, both in JAVA and in C, as anyone who tried naming a programming entity—be it a variable, a function, or a newly introduced type—must have noticed: the problem is not of finding the needle in the haystack, but the simple truth that, no matter how large the universe of discourse is, the competition on the few scarce good names remains fierce.

Striking a balance between the desire to make names descriptive and meaningful, and the practical demand that these are not overly verbose, we often wish to use identifiers such as print, close, sort, execute or draw in reference to distinct entities. Program blocks and scoping rules serve this wish in making it possible to reuse a name in different contexts in an orderly fashion. A common, yet controversial mechanism for reusing a name within the same context, is overloading, an ad-hoc kind of polymorphism [20].

Several style guides \(^1\) all but completely forbid the use of overloading. This practice could be justified, e.g., by the vigorous criticism by B. Meyer [95], expressed succinctly with his, almost axiomatically-true, statement:

| Different Things Should have Different Names |

But, this statement could be (and often is) answered by an equally self-evident truth

| The Same Things Should Have the Same Name |

which reveals the clumsiness in the encoding function signatures into their names, e.g., in the definition of a series of functions:

- `printInt(int i),`
- `printBoolean(boolean b),`
- `printChar(char c),` etc.,

\(^1\)http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
instead of straightforward use of overloading: `print(int i), print(boolean b), print(char c), etc.`

Meyer and others [13] point a finger at the ambiguity innate in overloading—an ambiguity which is exacerbated in the presence of inheritance, genericity, coercion, and language-specific mechanisms (e.g., non-explicit, single parameter constructors in C++, covariance in EIFFEL [67], etc.). Arguably, setting the rules for resolving this ambiguity may require a hefty load of language legalese, and a not so pleasant challenge to the unsuspecting programmer. Suffice to say that even the semantics of the trivial case of overriding one of two overloaded versions of a function is different in JAVA and in C++.

Constructors pinpoint the difference in opinion between the parties to this debate: JAVA, C++ and C# programmers are not free to name constructors as they please—all constructors of a given class must bear its name. Since constructors are not inherited, at least the intricacies of interaction between overloading and inheritance are saved. Still, even supporters may see flaws in constructor overloading. To quote a JAVAWorld article:2

"*With JAVA, the language design for constructors is quite elegant—so elegant, in fact, that it’s tempting to provide a host of overloaded constructors. When the number of configuration parameters for a component is large, there can be a combinatorial explosion in constructors, ultimately leading to a malady known as constructor madness... .”*

### 3.2 Studying The Use of Overloading

In this work, we contribute to the discussion between proponents and opponents of overloading by a study of the use of overloading in JAVA programs. For this study, we developed a taxonomy of categories (which can also be called patterns and even micro-patterns [46]), for the classification of the use of overloading, based mostly on the type of interaction between overloaded methods. This taxonomy is also characterized by stretching a spectrum of the use of overloading, from ad hoc patterns, in which overloading is coincidental, to systematic patterns, in which overloaded methods are semantically cohesive.

In order to estimate the prevalence of the various overloading patterns in actual code we conducted an empirical evaluation, in which we applied this taxonomy to

---

2“Java Tip 63, Jerry Smith, Nov. 1, 1998"
a large corpus of JAVA applications using a new research method. This method includes randomly sampling the corpus, manually evaluating the sampled items and testing the reliability of this evaluation, while employing techniques traditionally used in social sciences. Also of interest is the way in which the development of the taxonomy was in tandem with the two batches of work by human raters, and how the reliability of the human classification was estimated. This research method, to the best of our knowledge, was not previously applied to the study of software.

In the empirical evaluation we sought to answer the following questions:

1. What is the probability that a method, selected at random from the corpus, is overloaded?

2. What is the probability that a constructor, selected at random from the corpus, is overloaded?

3. For each of the overloading pattern, what is the probability that a method (or a constructor), selected at random, follows this pattern?

The answers to these questions provide evidence that overloading is used extensively in Java programs, and that, in contrast with the predictions of its opponents, overloading is used mostly in a systematic fashion.

The use of overloaded functions to implement a similar, but slightly different semantics, does prove that programmers do not abuse the mechanism. At the same time, even systematic use of overloading is not so desired from a software engineering standpoint. For the class’s author, this means a blown up interface with extra code to document and maintain. For the class’s client, this practice requires familiarity with different versions of essentially the same method.

Moreover, the semantics of the interaction between overloading and overriding varies between languages [13]. Understanding this subtlety is required in order to make sure that the intended method is indeed invoked. The example in Figure 3.2.1, drawn from [13], illustrates the problem.

Class Down presented in this figure overloads methods $f$ and $g$ introduced in its super class. Now, consider the following invocations:

\[
\begin{align*}
&\text{\texttt{(new Down()).\texttt{f(new Top())};}} \\
&\text{\texttt{(new Down()).\texttt{g(new Bottom())};}}
\end{align*}
\]

Which methods get called? The answer depends on the language in which this model is implemented. In JAVA, both calls invoke Up’s methods, while in C++ the first call results in an error and the second invokes Down’s $g$. The reason for these
Fig. 3.2.1 Different behavior in different languages

class Top{}
class Middle extends Top{}
class Bottom extends Middle{}

class Up{
    void f(Top t) {/*...*/
    void g(Bottom b) {/*...*/
}
class Down extends Up{
    void f(Middle m) {/*...*/
    void g(Middle m) {/*...*/
}

differences is that in C++, Down’s methods hide those of Up rather than overload them.

3.2.1 On Empirical Study of Programming Languages

The design of an object oriented programming language, just as an extension of one, is an art in many ways. In other ways, it is an exact science, requiring rigorous analysis of semantics, soundness, etc. and of course, exciting engineering is also involved. But, do we really understand how this tool is really used, or abused?

Both issues of data gathering and data analysis are what makes it difficult to understand how the industry really uses a programming language. But, these difficulties should not stop us from trying.

This work offers, in a sense, one direction at which such understanding may be gained. First, it uses Qualitas corpus\(^3\), an organized collection of software systems intended to be used for empirical studies in software engineering. Observing the size and the increasing acceptance of this corpus we can say that we are getting closer to a meaningful sample of the global concrete use of JAVA.

The issue of data analysis remains. Exact static analysis techniques are prohibitively resource consuming, especially when applied to such a large corpus. More importantly, for our purposes, we need a classification which is conceptual rather than syntactic—taking into consideration not only strictly adherence to a formally defined category but also close resemblance. For example, a method which

\(^3\)http://www.cs.auckland.ac.nz/~ewan/corpus/
invokes another, can be rewritten without such invocation, by simple inlining and then applying local polishing. It requires a human to reveal the fact that this inlined call is in fact a case of (say) default arguments.

(On a side note, recall that dynamic analysis is not easier than static analysis. It is a great achievement to assemble a software corpus from so many components. But, it is a much higher mountain to set out a running environment for all of these components, each with its own bugs, idiosyncratic reliance on external libraries of very specific versions, and specific weird constraints on the execution environment. And, as if this is not sufficiently difficult, the question of finding “typical” inputs or “runs” has to be addressed.)

The alternative direction taken here is of a controlled human evaluation. We somewhat compromise the preciseness of the definitions, and employ humans to classify and understand the studied body of software.

Of course, it is unrealistic to apply such human analysis to large data corpus such as Qualitas. But, it is possible, as we did here, to subject a random sample drawn from the corpus to human analysis and then use statistical methods to reason about the reliability of this analysis, and to deduce conclusions on the entire corpus, and through this, on the illusive global programming practice.

### 3.3 Research Method

This section describes the method of experimentation, both automatic and manual, and the Java corpus in which it was carried out.

#### 3.3.1 Definitions

The Java Language Specification [51] defines method overloading as follows:

> “If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, then the method name is said to be overloaded.”

Thus, overloading can occur between public and private methods, static and not-static methods, abstract and final methods, etc.

1. We restrict our attention to method (and constructor) overloading, even though one may argue that there are other kinds of overloading, e.g., when a class
features a data member and a function member of the same name. Similarly, we exclude overloading of the ’+’ operator, the final keyword, etc.

2. Even though the JAVA semantics precludes a definition of two methods which are different only in their return type, cases of this sort can be (and indeed are) found in .class files, e.g., as a means for implementing co-variance by certain JAVA compilers. This synthetic overloading is ignored in our study.

A constructor cohort is the set of constructors of a class. Methods are grouped in method cohorts, each being the maximal set of methods sharing the same name, and available in the same user-defined type, that is a class, an interface, an enum or an annotation. In this work we restrict attention to non-degenerate cohorts, i.e., cohorts with two or more peers.

The primary methods in a method cohort, are those which are first introduced or reimplemented in the type. The remaining methods, i.e., those which are inherited from a parent, are called secondary.

3.3.2 Data and Automatic Analysis

Our study started with the Qualitas corpus, consisting of 99 open-source JAVA applications, which was used extensively in the literature (e.g., in [8, 10, 91, 124, 142]) and considered to be well representing the standard programming practice in JAVA. The corpus was pruned to the most recent version of each application in it. Overall, the remaining data set consisted of 6,538 packages, with 128,482 classes, 14,214 interfaces, 48 enumerated types, and 106 annotations.

Our evaluation began with an automated analysis, which scanned the entire corpus for occurrences of overloaded methods and constructors. This analysis was carried out on the bytecode representation with a precise implementation of the above definition of overloading in the Java Tools Language (JTL) [26].

Constructor Cohorts

There were 162,495 constructors in total in the corpus. This number includes also synthetic constructors, which are generated automatically by the compiler when the programmer does not define any constructor for a given class, except when the class is anonymous, which do not have any constructors.

Figure 3.3.1(a), depicting the number of classes defining each number of constructors, shows the typical Zipf law distribution [144], that is \( f(k) \propto k^{-\alpha} \) (\( \alpha > 1 \),
as found in many software metrics [25]. It can be seen that the majority of all classes have only one constructor. (the exact number is 83%). We also found that 35% of constructors take part in non-degenerate cohorts. (Note that synthetic constructors never participate in non-degenerate cohorts.)

It is also evident there are a number of classes with a large number of constructors, and even two classes with as many as 25 constructors. Yet, on average the number of constructors per class is small: 1.264.

In the corpus, we found 1,030,623 method definitions (a definition of a method as abstract or in an interface being counted); this number includes redefinitions of methods. Of these, 148,192 methods are peers in 45,352 non-degenerate cohorts, i.e., slightly over 14% of the methods are overloaded. It follows that, as might be expected, overloading is much more prevalent among constructors than with methods.

**Method Cohorts**

Figure 3.3.1(b) is the equivalent of Figure 3.3.1(a), but focusing on method cohorts. It can be inferred from the graph that method cohorts tend to be larger than constructor cohorts. In fact, the average non-degenerate method cohort size is 3.27.

The linear decrease, typical of Zipf distribution, is not as evident here. With some imaginative effort, we can discern here a Zipf like distribution describing most cohorts’ sizes (note that the slope of the Zipf decrease in constructors is much shallower than in methods), combined with a cluster of giant cohorts with over 80
Clearly, the size of this cluster exceeds what might be predicted by the Zipf distribution. A closer look at the 304 cohorts with 80 methods or more, shows that almost all of these are part of an implementation of the VISITOR design pattern [42]. In fact, the name of 268 giant cohorts is simply visit, while 31 cohorts are named endVisit. The remaining 5 giants can probably be explained by the tail of the Zipf distribution.

3.3.3 Sampling and Human Classification

Pre-Test

The pre-test phase was designed to produce a taxonomy of overloading, consisting of clear and unequivocal definitions, which are not merely that, but also effective for classifying concrete use of overloading in JAVA.

The first draft of the taxonomy was written based on our own JAVA programming experience and on sporadic inspections of cohorts found in the corpus.

This draft was then perfected using the following process: A random sample of 100 method cohorts was selected from the ensemble of such cohorts found by the automatic analysis of the corpus. The sample was restricted to cohorts satisfying the following conditions: (i) The cohort is associated with a class. That is, we excluded cohorts of interfaces (no cohorts were found in enums nor in annotations). (ii) At least one method in the cohort was non abstract. Cohorts were then further trimmed down to include only methods defined in the same class, i.e., primary overloading.

The sample was then subjected to human classification as follows. First, we classified all cohorts, using the taxonomy draft. In the course of doing so, the taxonomy was refined, definitions were clarified, categories reorganized, etc. The refined taxonomy was then explained to three volunteer computer science graduate students, with a solid background in object oriented languages. (This explanation involved examples taken from the corpus, but not from the sample.) The raters were then asked to classify 50 specimens of the sample, 25 of which were common to all raters, while the remaining 25 specimens were specific to the rater. (Other than these conditions, the distribution of specimens among the raters was random.)

We then manually inspected the results, using discrepancies for further refinement of the definitions and the taxonomy.
Method Cohorts Categorization

Having gained our initial confidence in our taxonomy we proceeded to experimentation concerning reliability—that is the extent at which human rating according to it is reproducible. To do so, we repeated the rating of method cohorts. This time, with 10 recruits, undergraduate- junior and senior students. All students have successfully completed the Technion’s Object Oriented Programming course. They were each offered a monetary reward for their efforts (200 NIS, roughly equivalent to 50USD). The taxonomy and the categories in it were then explained to the raters in a two-hour frontal presentation (the presented slides are available online).

A newly selected sample of 100 method cohorts was then distributed among the participants, where the random distribution satisfied the conditions that each rater was assigned 40 cohorts and that each cohort was rated by four independent raters. To encourage seriousness, raters were promised (and paid) 2.5NIS (about $0.62) for each correct categorization. The rating process lasted about 2.5 hours. It was carried out in a supervised setting, in which the raters could not communicate with each other. In addition, and independent of the student raters, we also rated all cohorts.

A battery of statistical tests was then applied to the raw results of method cohorts classification. As reported below, these tests indicated that our rating of cohorts is reliable with high confidence margins.

Constructor Cohort Classification

Relying on the reliability of our classifications, we did not repeat the same process for constructor cohorts classification, instead, this classification was done solely by us. The sample consisted again of 100 cohorts selected at random from the constructor cohort base.

3.4 Taxonomy of Overloading

We now present the fruit of the experiments and process of perfecting a taxonomy of the use of overloading in JAVA. This section gives a high level survey of the main categories. The next section elaborates, describing the specific patterns in greater detail.

http://www.cs.technion.ac.il/~ssdl/pub/JavaMethodClassification/JavaOverloadingClassification.pdf
The primary question that our classification asks in considering an overloading incidence is how coincidental it is. An extreme case is, for example, a class representing a cartoon cowboy, featuring an overload of method `draw`. At the other end, will find, e.g., the overloaded method `setLocation` of class `awt.Point`, whose partial view is presented in Figure 3.4.1.

Fig. 3.4.1 An example of INTRINSIC overloading

```java
class Point {
    public int x;
    public int y;

    public void setLocation(Point p) {
        setLocation(p.x, p.y);
    }

    public void setLocation(int x, int y) {
        move(x, y);
    }

    public void move(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

A classification according to this criterion is important since overloading is criticized precisely because there is no enforcement of any semantical relationship between methods in the same cohort. Yet, in practice we find that such methods are often related, and that overloading is often used to capture a situation in which the input to a certain operation can be presented in different ways. Figure 3.4.2 draws a spectrum of the relationship between the semantics of overloaded methods.

The boxes in the figure represent overloading categories (which we will interchangeably also call patterns).

As we move to the right from the central dividing line we meet patterns which are progressively more systematic, that is, patterns in which the semantics of peers is progressively more related. Conversely, a move to the left reveals patterns in which overloading is of a more ad hoc nature. Boxes on the central line represent “neutral” patterns, i.e., patterns in which overloading can be either ad hoc or systematic.
Systematic overloading occurs, e.g., when the body of one overloaded method is in essence a transformation of its arguments, followed by a call to one of its cohort peers. Cases of this sort fall into the INTRINSIC category (this category includes also other kinds of overloading, as explained later). The POTENTIAL category is similar in that our human reviewers concluded that it can be brought into the INTRINSIC category with minimal effort. PEER-CALLERS are overloading instances in which a method calls its peer, but it is not clear whether it can be rewritten in the INTRINSIC form.

At the other end of the spectrum, the ACCIDENTAL category, refers to cases in which no peer calls occurred, and no other relationship between peers could be identified.

On the dividing line, we find, PLACEHOLDERS in which all methods in the cohort have no body. The overloading kind can fall into any other category, depending on the implementation in the inheriting class or classes. Cohorts of this patterns were excused from the sample since their classification is trivial. On this line, we also find the rather rare DUMMY ARGUMENT in which an extra, otherwise unused, argument distinguishes between peers (particularly constructors) which need the same arguments’ type sequence. (Think for example on distinguishing between a polar- and cartesian- based constructors to a class Point).

Notice that the above categories apply to a pair of peers. Different pairs selected from the same cohort, do not necessarily fall into the same category. An exception is the VISITORS category, which represents the use of overloading for realizing the VISITOR design pattern. Usually, in a cohort which is classified into the VISITORS category, most, if not all, peers fall into this category.

Finally, the PSEUDO GENERIC category pertains to cases of use of overloading in JAVA which were candidates to generic based implementation, had JAVA gen-

**Fig. 3.4.2** The overloading spectrum, from systematic to ad hoc.
ics been applicable to primitive types, e.g., as in the different implementations of Math.round for types double and float. Here again, we may expect several peers to fall into this category.

### 3.5 Overloading Patterns Catalog

In this section we discuss the overloading patterns in greater detail and exemplify their use. Our presentation starts from the systematic end of the overloading spectrum and progresses towards the ad-hoc patterns.

#### 3.5.1 INTRINSIC Overloading

The INTRINSIC category refers to methods whose relationship with its name-peer is semantical. Further breakdown of this category is offered by Figure 3.5.1.

**Fig. 3.5.1 Classification of intrinsic overloading patterns.**

In the figure we see that there are two main subcategories here: RESENDING in which defines an asymmetric relation between two methods in a cohort and INDUCED applies equally to all of the methods in a cohort.

**Induced Overloading**

In the INDUCED category, overloading in one cohort induces an overloading in another cohort. Suppose that the designer of a certain class sees it fit to equip this class with two constructors. Now, if this class is extended by way of inheritance, then it is only natural that the subclass will offer two constructors, each delegating to a distinct constructor in the base class. This situation occurs in many other situations, e.g., in design patterns [42] such as COMPOSITE and DECORATOR, and in general, in all cases in which a class delegates duties to another. In all of these,
the desire to provide a rich and consistent interface brings about overloading (in
the delegator) which replicates overloading in another cohort (the delegate).

The requirement in our natural language description of this category was dou-
ble folded: (i) the delegator and its delegate have identical argument list; (ii) the
deleator invokes the delegate precisely once in any of its execution paths.

The left hand side of Figure 3.5.1, shows a breakdown of INDUCED based on
the relationship between the delegator and the delegate:

1. **INTRA-CLASS DELEGATION**, in which each method invokes a method with
a different name but same arguments of the same class, and in the case the
callee is not static, the call is to this. One example of this pattern is the
cohort named removeBundle in Eclipse’s class RequireBundleHeader
located in package org.eclipse.pde.internal.core.text.bundle:

```java
void removeBundle(String id) {
    removeManifestElement(id);
}

void removeBundle(RequireBundleObject bundle) {
    removeManifestElement(bundle);
}
```

2. **SUPER-CLASS DELEGATION** type in which each method invokes a method
with the same signature on the super class. This pattern is common in con-
structors, as can be found in class RuntimeException of the JAVA standard
library:

```java
public class RuntimeException extends Exception {
    public RuntimeException() {
        super();
    }

    public RuntimeException(String message) {
        super(message);
    }

    public RuntimeException(String message, Throwable cause) {
        super(message, cause);
    }
```
public RuntimeException( Throwable cause) {
    super(cause);
}

3. INTER-CLASS DELEGATION, in which each method invokes a method with the same signature on a member object. The cohort named updateString in class AS400JDBCRowSet (found in package com.ibm.as400.access) drawn from the open source version of the IBM toolbox for Java (JTOpen) demonstrates this pattern:

```java
public class AS400JDBCRowSet
    implements RowSet, Serializable {
    /*... */
    private AS400JDBCResultSet resultSet_;  
    /*... */
    void updateString
        (int columnIndex, String columnValue) {
        validateResultSet();
        resultSet_.updateString(columnIndex, columnValue);
        eventSupport_.fireRowChanged(new RowSetEvent(this));
    }
    void updateString
        (String columnName, String columnValue) {
        validateResultSet();
        resultSet_.updateString(columnName, columnValue);
        eventSupport_.fireRowChanged(new RowSetEvent(this));
    }
}
```

**Resending**

In the RESENDING category, one overloaded method carries out its mission by resending its arguments to its peer after some preprocessing phase. We say that a designated caller method is RESENDING to a designated callee method when all four of the following conditions hold: (i) the caller invokes the callee precisely once in any of its execution paths, or there is a single call site, which is executed iteratively; (ii) the caller does not call any other peer; (iii) the returned type of the caller and the callee is the same; and (iv) if the caller returns a value, it is the value returned by callee, unaltered.
Figure 3.5.1 distinguishes between five patterns of RESENDING, based on the processing work carried out by the caller on the arguments it passes on to the callee:

1. PACKING, in which the caller packs some of its arguments into a collection or an array and then sends it to the callee. Method `setValue` of class `PreferenceConverter` of found in package `org.eclipse.jface.preference` of the Eclipse development environment, illustrates this pattern:

```java
void setValue (IPreferenceStore store, String name, FontData value){
    setValue(store, name, new FontData[] { value });
}
```

2. UNPACKING, in which the caller accepts a collection or an array, and invokes the callee on each element of the collection (array). One example of this pattern is method `convertToVector` of class `DefaultTableModel` which is located in package `javax.swing.table`:

```java
Vector convertToVector(Object[][] anArray) {
    if (anArray == null)
        return null;
    Vector v = new Vector(anArray.length);
    for (int i=0; i < anArray.length; i++)
        v.addElement(convertToVector(anArray[i]));
    return v;
}
```

3. CONVERSION, in which the caller converts one or more of its arguments to another type, to make it suitable for the callee to digest. Method `setLocation` of class `Point` depicted in Figure 3.4.1 is a case of this pattern.

4. REDUCING, in which the caller processes some of its arguments and sends a subset of the arguments to the callee, as is demonstrated by the create method of class `BidiOrder` which resides in package `com.ibm.as400.access` of the Azureus application:

```java
ResourceDownloader create(URL url, boolean force_no_proxy) {
    ResourceDownloader rd = create(url);
    if (force_no_proxy && rd instanceof ResourceDownloaderURLImpl)
        ((ResourceDownloaderURLImpl)rd).setForceNoProxy(force_no_proxy);
    return rd;
}
```
5. **Default Arguments** in which overloading is used as a substitute to default arguments mechanism, and the caller does nothing but resend all of its arguments, as well as some other default value or values, to the callee. The following constructors, which belong to class `Point`, presented in Figure 3.4.1 fall into this category:

```java
public Point() {
    this(0, 0);
}

public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
```

### 3.5.2 Potential Overloading

The refinement of the *Intrinsic* category as presented in Figure 3.5.1 is applicable in principal also to the *Potential* category. However, the manual task of identifying "Potential Induce" overloading, that is, checking whether a certain method pair *could be rewritten* by delegation to another such pair, which could be anywhere in the system, is formidable. We did not ask our raters to do that, and instructed them to concentrate in finding cases in which one method could be rewritten in terms of a name-peer, whereby restricting the breakdown of the *Potential* category into the various sorts of RESENDING.

For example, the first version of `setLocation` method in Figure 3.4.1, rewritten as

```java
public double setLocation(Point p){
    move(p.x, p.y);
}
```

belongs to this category.

### 3.5.3 Peer-Caller Overloading

A method is classified as *Peer-Caller* when it invokes one of its peers, but it is not classified as RESENDING.

Method `findResources`, drawn from class `StandardPluginClassLoader`, which resides in package `org.java.plugin.standard` illustrates such a case:

```java
public Enumeration findResources(final String name) {
```
List result = new LinkedList();
findResources(result, name, this, null);
return Collections.enumeration(result);
}

Since the return type of this method differs from that of the invoked peer it cannot be rewritten as an instance of the INTRINSIC category.

3.5.4 VISITOR Overloading

The VISITOR design pattern is a way of separating operations from the data structure upon which they operate. This pattern is often realized in JAVA by an interface which has a visit() method for each class whose objects may reside in the data structure. Thus, this design pattern usually implies using overloading. Moreover, often the data structure may contain objects of many different type, and as a result the number of overloaded visit() methods becomes very high.

One example of this category is the cohort named visit of class GenericVisitor, found in package org.eclipse.jdt.internal.core.dom.rewrite of the Eclipse framework. There are 83 methods in this cohort, each accept a single parameter, which represents a type of a node in the Abstract Syntax Tree representation of a JAVA program.

3.6 Statistical Analysis of the Experiment

The definitions of the various categories in the previous section may seem more intuitive than precise, and difficult to formalize. Take for example the definition of the DEFAULT ARGUMENTS category, which required that the “caller does nothing but resend all of its arguments”. Then, it is not difficult to construct spiteful cases, which challenge the accuracy and even decidability of, say, the phrase, “does nothing”—by adding involved computation, which would take tremendous resources for a statistical analyzer to prove vacuous, or by throwing in a reduction of the halting problem.

Instead of doing so, we restrict our classification to the intuitive meaning, as interpreted by humans, and devote this section to the reliability of this classification. We start with some general figures. Recall that raters where rewarded based on their hit rate, which was computed against the independent rating carried out by us. Scores ranged between 73% and 85%, and averaged at 79%.
As expected, there were many disagreements regarding classifications in the somewhat loosely defined POTENTIAL category. It turned out that almost all disagreements were with regard to this category. In contrast, the DEFAULT ARGUMENTS category raised the fewest disagreements, reaching a fully unanimous vote casted in 85% of the cases.

The more important question which we explore next is the systematic statistical reliability of this human classification. The analysis here shall demonstrate that the results of the manual classification are indeed reproducible, and that the numerical value that will be presented in the following section are therefore significant. At the end of the current section we remind the reader the notion of confidence interval, which should help in deducing conclusions regarding the entire corpus from what was observed for the sample.

### 3.6.1 Reliability of Human Classification of Overloading

*Cronbach’s α-coefficient* [31], or for short $\alpha$, is a statistic which is used in social sciences to estimate the internal consistency of multiple items within a scale. It is employed in cases, such as ours, in which the scale is nominal rather than ordinal or rational. The value of this estimator ranges between 0 and 1, where a value of 0 corresponds to the case that items are uncorrelated, i.e., all variation is due to random fluctuations. A value of 1 corresponds to the case that the items are in complete correspondence. For research purposes it is customary to require $\alpha \geq 0.8$ [101].

*Cohen’s κ-coefficient* [24], or for short $\kappa$, is a leading measure of agreement which assesses the extent to which two raters give the same ratings to the same objects, while factoring out the probability of agreement between the raters that would be expected due to chance. The $\kappa$ values range from -1, which means perfect disagreement to 1, meaning perfect agreement, where 0 is interpreted as agreement achieved by chance. A value of 0.6 or higher is considered as a strong agreement.

Table 3.1 presents the values of these two statistics in relation to the classifications carried out by the human raters in the experiment. The first row corresponds to evaluations of all cohorts which fell in the sample, while the second is restricted to cohort samples of size 2.

Examining the table, we see that the high hit rate the evaluators achieved is far from being accidental, and it cannot be attributed to chance. Further, we see high correspondence not only in classification according to top level categories, but also to sub categories.
Few words are in place in order to explain the method of computation, which involved data aggregation. Recall that ten raters participated in the experiment, each classifying a subset of 40 methods, while there was no single pair of raters who classified the same subset. However, calculating the $\alpha$ measure requires that all raters refer to the same items. Thus, instead of considering a model in which 10 raters evaluated 40 items each, we switched to a model in which we consider only the ratings of each item, without taking into consideration who rated it. Our transformed model therefore included four sets of ratings, each referring to 100 cohorts.

Although the $\alpha$ measure is not originally designed to check inter-rater reliability, there is evidence showing that its use for such purposes is adequate, and even desirable when multiple raters are involved [82]. Following transposing of the data set, this statistic was used to estimate the correspondence between all ratings, rather than raters.

We used the same aggregated model for $\kappa$ calculations as well. The $\kappa$ measure was used in this study to estimate inter-rater reliability between our rating and each of the aggregated rating sets. The table displays the average value of the four $\kappa$ values that were obtained.

Finally, we should say that the excellent values reported in Table 3.1 are relevant only to the categories which were actually presented in the sample. As we shall see below, some of the categories in the taxonomy, although theoretically interesting, did not manifest in the sample.

### 3.6.2 Binomial proportion confidence interval

Now that we have established the statistical significance of the manual classification, it remains to determine what can be inferred about the entire corpus from the classification of the specimens in the sample. Suppose that a fraction of size $p$ of
the elements in a sample fell into a certain category, then, we would like to find a value $\Delta p$ such that there is a vanishing probability that the true fraction of cases in the corpus is not within between $p - \Delta p$ and $p + \Delta p$.

The binomial proportion confidence interval provides this information precisely. It uses the proportion estimated in a statistical sample and allows for sampling error. There are several ways to compute a confidence interval for a binomial proportion. We chose the Wilson score interval \cite{136} due to the good properties of this test for even a small number of trials or an extreme probability. To estimate the sampling error we calculated 95\% confidence intervals using Wilson score method for a binomial proportion.

<table>
<thead>
<tr>
<th>Proportion in Sample</th>
<th>Confidence Interval $n = 77$</th>
<th>0%–6%</th>
<th>0%–8%</th>
<th>1%–13%</th>
<th>5%–19%</th>
<th>10%</th>
<th>20%</th>
<th>35%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence Interval</td>
<td>n = 77</td>
<td>0%–6%</td>
<td>0%–8%</td>
<td>1%–13%</td>
<td>5%–19%</td>
<td>10%</td>
<td>20%</td>
<td>35%</td>
<td>50%</td>
</tr>
<tr>
<td>Confidence Interval</td>
<td>n = 65</td>
<td>0%–8%</td>
<td>0%–9%</td>
<td>1%–15%</td>
<td>4%–20%</td>
<td>12%–30%</td>
<td>25%–46%</td>
<td>38%–62%</td>
<td></td>
</tr>
</tbody>
</table>

We calculated the confidence intervals based on a sample of size 77 (method cohorts of size two) and 65 (constructor cohorts of size two), for various proportions in the sample. The results are presented in Table 3.2. As can be seen in the table, the values of $\Delta p$ are quite large, but still, if a certain pattern is infrequent in the sample, it is with very high probability infrequent in the corpus. Conversely, patterns which are common in the sample, are very likely to be common in the corpus.

### 3.7 Results

Now that we have established the reliability of our classification system and understood what can be inferred from its values to the full corpus, it is time to present the actual results of this classification.

#### 3.7.1 Method Cohorts

Table 3.3 shows the distribution of sizes of cohorts that fell in the sample of 100 cohorts. As expected, a number of large cohorts were sampled. Even though the small cohorts, with only two methods, were 77\% of the samples, the methods in these were 64\% of the sample.
Tab. 3.3 Distribution of cohorts’ sizes in the 100 method cohorts sample

<table>
<thead>
<tr>
<th>Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cohorts</td>
<td>77</td>
<td>13</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td># Methods</td>
<td>154</td>
<td>39</td>
<td>24</td>
<td>5</td>
<td>12</td>
<td>7</td>
<td>241</td>
</tr>
<tr>
<td>Fraction</td>
<td>64%</td>
<td>16%</td>
<td>10%</td>
<td>2%</td>
<td>5%</td>
<td>3%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Tab. 3.4 Manual classification of the 100 method cohorts in the sample (zero values are omitted).

<table>
<thead>
<tr>
<th>Category</th>
<th>Sub-category</th>
<th>Sub-sub-category</th>
<th># Cohorts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intrinsic (68)</td>
<td>Resending (54)</td>
<td>Default arguments</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduction</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Packing</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unpacking</td>
<td>1</td>
</tr>
<tr>
<td>Induced (14)</td>
<td>Inter-class delegation</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intra-class delegation</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Super-class delegation</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Potential (18)</td>
<td>Resending (18)</td>
<td>Default arguments</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unpacking</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.4 provides the results of manual classification of these cohorts. The numbers in the table represent the results of our classification. Note that a cohort with more than two peers could fall into several categories. In the table, a cohort was counted in a certain category if the pattern occurred in it at least once. Categories whose number of occurrences is zero are omitted.

In addition to the counts depicted in the table, 2 cohorts were such that none of the methods were implemented, i.e., PLACEHOLDERS, 13 were classified as ACCIDENTAL since none of their methods invoked any of their cohort peers (or could be implemented as such), and 6 cohorts contained methods which invoke each other, but did not match any of the patterns and were therefore classified as PEER-CALLERS. The sample did not include any instances of DUMMY ARGUMENT, VISITOR and PSEUDO GENERIC.

The table reveals a strong tendency towards systematic rather than ad hoc use of overloading: More than half of the cohorts involve a resending pattern. The
most frequent pattern of overloading that was observed is that of default parameters, which was observed in 28% of the cohorts and has the potential of being implemented in additional 9%.

3.7.2 Method Pairs

The quadratic increase in the number of pairs of peers makes it difficult to analyze the patterns of use of overloading in cohorts with more than 2 methods. Worse, inspecting in isolation all possible pairs in a cohort is likely to produce confusing information which may need a bit of pondering before the underlying structure of the cohort can be revealed.

Consider for example the fill cohort in class Arrays, which has 18 methods (2 for each of JAVA’s primitive type and 2 for Object) and 153 different pairs. These pairs can be broken down as follows. The 9 pairs of the sort of
\[
\langle \text{fill(float[]),int,int,float} \rangle, \text{fill(float[]),float} \rangle
\]
are DEFAULT ARGUMENTS; (replacing float in any other primitive type or in Object). The 36 pairs of the sort of
\[
\langle \text{fill(float[],int,int,float)}, \text{fill(char[],int,int,char)} \rangle
\]
(where float and char can be replaced likewise) are PSEUDO GENERIC. And, the 36 pairs of the sort of
\[
\langle \text{fill(byte[],byte)}, \text{fill(Object[],Object)} \rangle
\]
are also are PSEUDO GENERIC. The remaining 72 pairs, which constitute 47% of the lot, do not fit into any of the categories.

Assigning appropriate weights to the different categories for cohorts with more than 2 peers can be complicated. But, even if this hurdle is overcome, requiring the human raters to reveal the underlying structure would not only have complicated the experiments, but also introduced unnecessary noise. Therefore, our more in depth analysis was restricted to the 77 cohorts in the sample which were of size 2. Table 3.5 provides a breakdown of the classification of these cohorts. Unlike Table 3.4, each cohort occurs precisely once in this table.

As in Table 3.4, the most common pattern is that of DEFAULT ARGUMENTS, being used by 24.6% of the cohorts, with additional 9% which have the potential of using it. Again, we see strong tendency towards systematic use of overloading. This tendency is further depicted visually in Figure 3.7.1, which portrays a histogram of the breakdown into top level categories.
Table 3.5 Results of manual classification of 77 method cohorts of size two (zero values are omitted).

<table>
<thead>
<tr>
<th>Kind</th>
<th>Sub-kind</th>
<th>Pattern</th>
<th># Cohorts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intrinsic</strong></td>
<td>57% (44)</td>
<td>Resending</td>
<td>44% (34)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default arguments</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduction</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Packing</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unpacking</td>
<td>1</td>
</tr>
<tr>
<td><strong>Induced</strong></td>
<td>13% (10)</td>
<td>Inter-class delegation</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intra-class delegation</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Super-class delegation</td>
<td>0</td>
</tr>
<tr>
<td><strong>Potential</strong></td>
<td>19% (15)</td>
<td>Resending</td>
<td>19% (15)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default arguments</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unpacking</td>
<td>1</td>
</tr>
<tr>
<td><strong>Accidental</strong></td>
<td>13% (10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Placeholders</strong></td>
<td>3% (2)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>Peer-Callers</strong></td>
<td>8% (6)</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

We see that 84% of the total weight falls in the right hand side of the figure. Six out of seven of the pairs in our sample exhibit systematic overloading, and three out of five fall in the **INTRINSIC** category. Also, summing up the values in Table 3.5 we determine that in three out of five pairs of overloaded methods, one method calls another. Relying on the confidence intervals summarized in Table 3.2, we can further infer that with high probability these estimates apply to the full corpus, to within a ±10% margin.

Finally, we remark that the tendency towards more systematic use of overloading should increase, or at the least stay the same, as the cohort size increases. This is of course with the exception of visitors, in which even though the intended semantics is similar, it is unclear whether the actual implementation of different visitors is likely to show systematic repetition. Luckily, visitors are very rare, and we can therefore conclude that the use of overloading in the vast majority of cases is very systematic, and that programmers are not tempted to abuse this language feature.
Fig. 3.7.1 Spectrum of systematic overloading in sampled method cohorts of size 2.

3.7.3 Constructor Cohorts

Table 3.6 displays the results of manual classification of the 100 cohorts in the constructors sample.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Sub-kind</th>
<th>Pattern</th>
<th># Cohorts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intrinsic</td>
<td>Resending</td>
<td>Default arguments</td>
<td>25</td>
</tr>
<tr>
<td>(59)</td>
<td></td>
<td>Conversion</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Packing</td>
<td>0</td>
</tr>
<tr>
<td>Induced</td>
<td>Resending</td>
<td>Inter-class delegation</td>
<td>0</td>
</tr>
<tr>
<td>(25)</td>
<td></td>
<td>Intra-class delegation</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Super-class delegation</td>
<td>22</td>
</tr>
<tr>
<td>Potential</td>
<td>Resending</td>
<td>Default arguments</td>
<td>12</td>
</tr>
<tr>
<td>(13)</td>
<td></td>
<td>Conversion</td>
<td>1</td>
</tr>
</tbody>
</table>

As in Table 3.4, we did not count each pair of overloaded constructors separately, instead, each cohort was counted once for each pattern that occurred at least
once. In 32 constructor cohorts no particular pattern was identified between any of the pairs, and hence, the entire cohort is classified as ACCIDENTAL.

No constructor cohort was classified as vanilla PEER-CALLER. In other words, there was no constructor cohort in which a constructor invokes other constructor and does not match a more specific pattern. This is not very surprising, since the language syntax mandates that the inter-constructor invocation must be the first statement. The only allowed computation, computing the actual arguments prior to the invocation, does not admit much programming freedom or creativity.

Examining the table further, we see a clear tendency towards more systematic use of overloading. But, in comparison with Table 3.4, it is evident that this tendency is not as forceful as it is with methods.

### 3.7.4 Constructor Pairs

In order to appreciate more accurately the tendency towards systematic overloading in constructors, we now concentrate, as we did with methods, in cohorts of size 2. Table 3.7 shows the distribution of sizes in the sample of constructor cohorts. We see that still, a substantial portion of the constructors fell in the first column of the table.

<table>
<thead>
<tr>
<th>Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cohorts</td>
<td>65</td>
<td>21</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td># Methods</td>
<td>130</td>
<td>63</td>
<td>36</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>8</td>
<td>262</td>
</tr>
<tr>
<td>Fraction</td>
<td>50%</td>
<td>24%</td>
<td>14%</td>
<td>2%</td>
<td>2%</td>
<td>5%</td>
<td>3%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3.8 presents a view of the results which contains classifications of constructor cohorts of size two.

Comparing the results of constructors classifications to those of methods classifications it can be observed that INDUCED overloading is more frequent in constructors than in methods. and that the one of the most common pattern is SUPER CLASS DELEGATION. Again, this is what may be predicted by the language syntax: a constructor with a non-empty parameters list in a base class often induces a constructor with the same parameters list in its derived classes.

Finally, Figure 3.7.2 summarizes the spectrum of use of overloading in restricted sample. The tendency towards systematic use of overloading is evident,
Tab. 3.8 Results of manual classification of 65 constructor cohorts of size two (zero values are omitted).

<table>
<thead>
<tr>
<th>Kind</th>
<th>Sub-kind</th>
<th>Pattern</th>
<th># Cohorts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intrinsic</strong></td>
<td>46% (30)</td>
<td>Resending</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default arguments</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduction</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Packing</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>20% (13)</td>
<td>Induced</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inter-class delegation</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intra-class delegation</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Super-class delegation</td>
<td>13</td>
</tr>
<tr>
<td><strong>Potential</strong></td>
<td>17% (11)</td>
<td>Resending</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default arguments</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion</td>
<td>1</td>
</tr>
<tr>
<td><strong>Accidental</strong></td>
<td>37% (24)</td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

Fig. 3.7.2 Spectrum of systematic overloading in sampled constructor cohorts of size 2.

but it is also clearly weaker than in methods (Figure 3.7.1).

We see that about three out of five cohorts with two constructors exhibit systematic overloading at some level, and about one in two is a resend to peer. Again,
we have the same reasons to believe that this tendency is not spoiled as we move to larger cohorts.

3.8 Conclusions and Further Research

3.8.1 Summary of Results

We found that overloading is used extensively in the corpus: 35% of all constructors and 14% of all methods. Cohorts tend to be larger in methods, with an average cohort size slightly greater than 3, while the average number of constructors per class is about 1.3. The distribution of cohort sizes is Zipf-like, except that method cohorts feature cluster of large cohorts attributed to the VISITOR design pattern.

We developed a taxonomy for the classification of the use of overloading in actual JAVA programs. The taxonomy was refined and exacted in a process involving two stages of subjecting sized samples to human rating in a controlled environment. The reliability of the classification was validated, at least for the major categories, by engaging statistical tests traditionally used in social sciences.

Statistical analysis also showed that at least six out of seven cases of use of method overloading are more systematic than ad hoc. The fact that overloading is mandatory in the definition of multiple constructors probably explains our finding that the systematic overloading is somewhat less frequent in constructors, occurring in about three out of five cases. These results (whose error margin is about 10%) may answer the allegation that overloading is likely to be abused.

It was determined that the most frequent use of overloading is for simulating defaults arguments. This use pattern occurring in about a quarter of overloaded methods, while additional 10% of these can probably be rewritten as such. Similarly, about a third of the cases in which overloading is used with constructors, are, or can be, expressed as a form of overloading.

3.8.2 Further Research

It is interesting to study the smaller categories which were not captured by our sample. This can be done, e.g., by employing refined sampling techniques.

We are intrigued by the INDUCED category, which suggests that overloading is viral—the use of overloading in one class leading to overloading in another. There may be room for checking whether generics could address this duplication.
JAVA does not support operator overloading, since it is considered potentially unsafe, as articulated by James Gosling [119]: "I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++." It will be interesting to study the use of operator overloading in C++.
Chapter 4

Keyword- and Default-Parameters in JAVA

Overloading is a highly controversial programming language mechanism by which different methods of the same class are allowed to bear the same name. Despite the criticism, JAVA programmers make extensive use of this mechanism—not just because it is available, but also because the language does not provide an alternative for defining multiple constructors, and because it is useful for expressing similarity of services provided by a class.

As we argued in the previous chapter, more than 60% of the overloading cases are “justifiable” and in 35% of the cases overloading is used for emulating a default arguments mechanism.

In this chapter we discuss two language features: keyword arguments and default arguments, and argue that their use in tandem reduces the use of overloading and enhances readability and maintainability of the code. Attaching names to parameters in the invocation command highlights their meaning, and reduces the need to examine lengthy documentation. The combination of keyword parameters and default values allows one to specify the actual parameters in any order while omitting any subset of the parameters that have default values.

This combination is used in several programming languages including LISP [53] and PYTHON [81]. C# designers recognized the advantages of the combination, and a mechanism that supported named and optional parameters was added to version 4.0 of the language\(^1\).

\(^1\)http://msdn.microsoft.com/en-us/library/dd264739.aspx
After arguing for the merits of this combination, we describe our JAVA language extension supporting it. The extended compiler is available for download at: http://ssdl-wiki.cs.technion.ac.il/wiki/index.php?title=Call_by_Name_Java_Extension. With the extension, every subset of the formal parameters may be assigned default values. A default value is either a constant or an expression involving method calls, data members and other formal arguments, which is evaluated at the scope of the declared method. The dependency between the formal arguments may even be circular, under some constraints.

Our extension allows a programmer to supply actual arguments either (i) **positionally**, i.e., in the order of the formal parameters, (ii) in a **keyword based** fashion, where each argument is preceded by the name of the formal parameter, or, (iii) in a **mixed style** by which a prefix of the actual arguments is specified positionally, while the remainder is specified by name. We identify the difficulties of interaction of this mechanism with inheritance, and explain how to deal with these while preserving the compatibility principle of Meyer [93] and Liskov [80].

**Chapter Outline.** The remainder of this chapter is organized as follows. Section 4.1 motivates the use of keyword arguments with default values as a substitute for overloading. Section 4.2 describes how the distinction between operands and options is supported using keyword and default arguments. Section 4.3 presents a JAVA language extension that supports keyword and default arguments and describes its implementation. Section 4.4 surveys related work, highlighting the arguments against and for keyword arguments. Section 4.5 concludes.

### 4.1 Solving the hardship of overloading in JAVA

In this section we emphasize the advantages of keyword parameters and default values over the practice of using overloading.

Unlike C++, C# and ADA [120], methods in JAVA cannot declare default values for their formal parameters. Overloading is available as a substitute: A default argument can be emulated by introducing a new method bearing the same name which invokes the original with an appropriate value for this argument. For example, method `getInteger(String)` in the JAVA’s standard `Integer` class, supplies a default argument to the more general `getInteger(String, Integer)` method:

```java
public static Integer getInteger(String nm) {
    return getInteger(nm, null);
}
```
For the class’s author, this emulation of default parameters means a blown up interface with extra code to document and maintain. In this example, the three-lined function `getInteger(String)` incurs a 28-lines documentation overhead toll. For the class’s client, this practice requires familiarity with different versions of essentially the same method, and understanding of the subtle semantics of overloading, and its not so trivial interaction with overriding [13], in order to make sure that the intended method is indeed invoked.

Overloading is a highly controversial language construct [95]. When used correctly it allows to capture similarity between different methods and to emphasize the fact that several different methods represent the same conceptual operation. However, it has the potential of being abused, by assigning the same name to conceptually different methods. As a result, several style guides ² all but completely forbid the use of overloading.

As we argued in the previous chapter, more than 60% of the overloading cases are “justifiable” and in 35% of the cases overloading is used for emulating a default arguments mechanism.

Having stressed that a keyword parameters and default values mechanism could replace a substantial portion of the cases of overloading, we now discuss the benefits of such a mechanism. Consider for example JAVA class `Point` depicted in Figure 4.1.1.

Fig. 4.1.1 A JAVA class representing a two-dimensional point.

```java
class Point {
    private int x, y;

    public Point(int x, int y) { this.x = x; this.y = y; }
    public Point() { this(0, 0); }
    //...
}
```

In the example, variables `x` and `y` are optional and have 0 as default values. To realize this, the constructor is defined twice and the second definition passes the default values to the first, which does the actual work. In JAVA, overloading is mandatory in the definition of multiple constructors, since programmers are not free to name constructors as they please— all constructors of a given class must

²http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
bear its name. In a typical class, there are many constructors that forward their work to each other, and to super-classes as well. For example, in order to support the option that the y parameter is optional, another constructor has to be added. Such a constructor is likely to be implemented by invoking the first constructor of Figure 4.1.1.

This is a cumbersome solution for the simple problem of realizing an optional default value. For each possible combination of optional arguments, an additional method must be provided.

Named parameters together with default values, offer a simple substitute. The host of methods (or constructors) doing nothing other than forwarding to other methods of the same name, but still carrying the price tag of extra documentation, interface bloat, and maintenance issues, can be replaced by a single method with appropriate defaults.

This is the case, for example, with those constructors of the JRE’s standard implementation of class String in charge of creating a String object from a byte array.

Fig. 4.1.2 Overloading and forwarding as substitute to default arguments in the constructors of String.

```java
public String(byte bytes[]){
    this(bytes, 0, bytes.length);
}
public String(byte bytes[], int offset, int length){
    /*...*/
}
public String(byte bytes[], String charsetName){
    this(bytes, 0, bytes.length, charsetName);
}
public String(byte bytes[], String charsetName){
    /*...*/
}
public String(byte bytes[], Charset charset) {
    this(bytes, 0, bytes.length, charset);
}
public String(
    byte bytes[], int offset, int length, Charset charset){
    /*...*/
}
```
Figure 4.1.2 demonstrates the process of forwarding while supplying defaults among the six overloaded constructors: `String(byte[])`, the first constructor in the figure, gives default value to the offset and length parameters while calling the second constructor, whose signature is `String(byte[], int, int)`; the same two arguments are given the same default values in the call of the third constructor, `String(byte[], String)`, to the fourth constructor, whose signature is, `String(byte[], int, int, String)`, and in the call of the penultimate constructor, `String(byte[], int, Charset)`, to the last constructor, `String(byte[], int, int, Charset)`.

A calling mechanism featuring parameter defaults is more than just syntactic sugar for method overloading; it can deal with the situation in which several arguments are of the same type—a situation which baffles JAVA’s overloading mechanism. In class `Point` of Figure 4.1.1 for example, we see a constructor in which both `x` and `y` default to 0, but it is impossible to declare constructors for the situations in which either `x` or `y` are missing, by adding both

```java
Point(int x) { this(x,0); }
Point(int y) { this(0,y); }
```

to the set of constructors of class `Point`. These two constructors definitions are contradictory since plain JAVA uses parameter types (and these types only) for resolving overloading ambiguities. The situation is slightly better with methods, whose name can be changed to support a variety of argument combination. With the same `Point` example, an attempt to define a variety of `move` methods by writing, e.g.,

```java
move(int x) { /* ... */ }
move(int y) { /* ... */ }
move(int x, int y) { /* ... */ }
```

will be rejected by the compiler due to the overloading problem, but with methods (as opposed to constructors), the method name can be changed to circumvent the ambiguity hurdle:

```java
moveX(int x) { /* ... */ }
moveY(int y) { /* ... */ }
move(int x, int y) { /* ... */ }
```

Method renaming addresses ambiguity but fails to capture the similarity between the three varieties of `move`—each method must be documented, implemented and maintained separately. The alternative offered by default values and keyword pa-
rameters is depicted in Figure 4.1.3.

Fig. 4.1.3 An implementation of class Point with keyword parameters and default values.

```java
class Point {
    private int x, y;

    public Point(int x = 0, int y = 0) {
        this.x = x; this.y = y;
    }

    public void move(int x = 0, int y = 0) {
        this.x += x; this.y += y;
    }

    // ...
}
```

Note that in this case a default arguments mechanism, as in C++, is not enough. Such a mechanism does not provide any way to omit the x value. An elegant way to achieve this is by using a combination of keyword and default arguments.

Another overloading difficulty which finds a more elegant solution with named parameters is that of passing a null value. For example, Class String contains eight versions of the static method named `valueOf` which accept a single parameter. The parameter types are boolean, char, char[], double, float, int, long, and Object. The documentation of the `valueOf(Object)` method states that this method returns null if the passed in object is null. However, the call `String.valueOf(null);` surprisingly throws a `NullPointerException`. The reason for this is that this call invokes the version taking `char[]`, as this is the most specific method.

This difficulty in inferring the chosen method can be resolved by using keyword parameters, where the name of the null parameter is explicitly specified.

### 4.2 Support for operands and options

Recall the famous distinction between operands—the (usually few) values on which a subprogram operates and (the usually many) options—which set the mode of operation [92]. With a keyword argument calling scheme, the designer places operands first on the formal parameter list, allowing them to be called positionally. Options follow in an arbitrary order, with appropriate defaults (See also an
ADA style guide\(^3\) that makes recommendations in this spirit, without making the explicit distinction between operands and options.)

Consider for example a method \(m\) taking two operands and \(n\) options as depicted in Figure 4.2.1(a). With the suggested language extension, this method can

**Fig. 4.2.1** (a) definition of a method taking two operands and \(n\) options using plain JAVA syntax, (b) its rewrite with our language extension, and (c) an example of how this method might be called with this extension.

```java
class A {
    void m(
        String operand1,
        int operand2,
        O1 opt1, O2 opt2,...,
        On optN) {
        // ...
    }
}

(a)
```

```java
class A {
    void m(
        String operand1,
        int operand2,
        O1 opt1 = defaultExp1,
        ...
        On optN = defaultExpN) {
        // ...
    }
}

(b)
```

```java
new A().m(
    "Restaurant", 42,
    opt17 := E, opt3 := E');

(c)
```

be rewritten (Figure 4.2.1(b)) to highlight the distinction between operands and options.

Figure 4.2.1(c) demonstrates a call to method \(m\) with "Restaurant" and 42 as operands, while setting \(\text{opt17}\) to the expression \(E\) and \(\text{opt3}\) to the expression \(E'\).

The use of overloading would have required \(2^n\) versions of method \(m\); with each of these versions, there is a need to lift the burden of deciding on, and then remembering the order of parameters. And, in the example, it is not even clear that a call such as

\[
m("Restaurant", 42, E, E');
\]

would have carried enough information to pinpoint the correct overloaded version of \(m\).

The distinction between options and operands penetrated EIFFEL’s standard

library, which uses a variety of means [94] to avoid passing options as arguments to methods. These means include the placement of setters and getters for shared or per-object option fields within the class, passing values for options to the class constructor, and, in the case of boolean options, writing two distinct versions of the main operation. Figure 4.2.2(a) shows how class A, the class enclosing method m can be rewritten to include setters for each of the options that this method takes. Figure 4.2.2(b) demonstrates how these setters are used for setting the options.

![Fig. 4.2.2](a) defining setters for options for the method of Figure 4.2.1, and (b) using these in a concrete call equivalent to Figure 4.2.1(c).

```java
class A {
    void m(String operand1, int operand2) {
        // ...
        O1 opt1 = defaultExp1;
        ...
        On optN = defaultExpN;
        A setOption1(O1 value) {
            opt1 = value; return this;
        }
        ...
        A setOptionN(On value) {
            optN = value; return this;
        }
    }
}

new A()
    .setOption1(E).setOption3(E').m("Restaurant", 42);
```

Clearly, the version using our language extension is shorter and clearer, not only on the supplier side, but also, and more importantly, on the client side.\(^4\) Still, Eiffel’s approach is a viable alternative if a number of methods defined in a class share options, in which case, a client would only need to learn once how to use these options. In the case that settings of these options tend to be the same in distinct call sites, the Eiffel approach might be preferred.

\(^4\)Even the number of tokens in each call is smaller; the overhead in terms of token count of setting \(m\) options by the EIFFEL approach is \(4m\), compared to \(3m\) using the proposed language extension.

91
Fig. 4.2.3 (a) defining an arguments object class for the method of Figure 4.2.1, and (b) using it in a concrete call equivalent to Figure 4.2.1(c).

```java
class Options {
    O1 opt1 = defaultExp1;
    ...
    On optN = defaultExpN;
    Options setOption1(O1 value) {
        opt1 = value; return this;
    }
    ...
    Options setOptionN(On value) {
        optN = value; return this;
    }
}
```

(a)

```java
new A().m("Restaurant", 42,
    new Options()
        .setOption17(E)
        .setOption3(E')
);
```

(b)

An even more viable alternative in this case is of using arguments objects [100] as depicted in Figure 4.2.3(a). Figure 4.2.3(b) shows how an arguments object is used while calling this method. Again, the arguments object alternative is longer and not as direct as using a keyword arguments calling scheme.\(^5\)

### 4.3 A JAVA extension for keyword and default arguments

This section describes the design and implementation of a JAVA language extension to support an expressive, yet easy to use, defaults mechanism. Our extension features:

1. **Default arguments**—a denotation that a parameter is optional and for supplying a default value for it. This default value is either a constant or an

\(^5\)As indicated, e.g., by the \(5 + 4m\) token-count overhead for setting \(m\) options with the arguments object alternative.
expression, evaluated at the scope of the declared method. Such an expression may involve other parameters, methods and fields of the class in which the method was defined, and any other entities defined at the scope of the declared method.

2. **Keyword arguments**—a mechanism that allows clients to provide arguments in an invocation as name/value pairs. Because each argument is named, the arguments can be supplied in any order; the binding of actuals to formals is carried out automatically by the compiler.

3. **Extended invocation syntax**, in which the list of actual arguments to a method has two, possibly-empty, parts:

   (a) **Positional Arguments List**, in which arguments are supplied in the order they are defined, followed by

   (b) **Keyword-Based Arguments List**, in which each argument is prefixed with the name of the formal parameter.

   This invocation syntax makes it possible for an invocation style which is either entirely *positional*, entirely *keyword based*, or mixed.

In C++ a parameter with no default cannot follow a defaulted parameter, and, the default value’s expression can not use other formal parameters, non-static data members and functions. In our extension, just as in C++, the initialization expression is computed in the context of the declared method, however, it may involve any function calls and data members access, as long as these are recognized in this context. This makes default value expressions equivalent to full-blown methods, with the right scoping and late binding properties. Moreover, thanks to keyword-based invocation, in our defaults mechanism the initialization expression may involve values of other parameters to this method and the dependency may be circular. We may declare a method in an `Interval` class such as:

```java
public void setInterval(
    int left = right - width,
    int width = 0,
    int right = left + width) {
    // ...
}
```
where the left parameter depends on the right parameter and vice versa. All we require is that if certain parameters are omitted from an invocation, they could be computed from the supplied arguments.

The above definition supports a full positional specification of all parameters, 
\texttt{setInterval(1, 7, 8);}
just as a full keywords based call
\texttt{setInterval(width := 7, left := 1, right := 8);}
Both forms allow omitting arguments with defaults: In the partial positional based call
\texttt{setInterval(1, 7);}
the defaults mechanism will complete the other parameter, setting right to 8. Similarly, in the partial- keyword-based call
\texttt{setInterval(left := 1);}
the defaults engine will set width to zero and right to 1. The circular defaults dependency, i.e., having both left depend on right and right depend on left is never a problem. If one of these arguments is missing, its value is computed based on the other. It is however illegal to invoke \texttt{setInterval} while omitting both left and right.

The dependency relationships between parameters are static, determined at compile time. Thus, in the definition
\texttt{public void f(int a, int b, int c = (b != 0) ? a : -1) { \\
    //...
}}
the c parameter always depends on the a parameter, even though the a value may not be used in the computation of the c value.

### 4.3.1 Methods with Default Arguments

#### Method Declaration

For each method with default parameters, our modified JAVA compiler computes the set of calling patterns, that is, those subsets of the formal arguments from which the remaining arguments can be computed. Every calling pattern with the exception of the pattern including all parameters, is realized as an auxiliary method computing the remaining arguments and then invoking the original method. This method represents the calling pattern in which all arguments are specified. The
names of these auxiliary methods are a mangled encoding of (i) the name of the original method, (ii) the types of arguments to the original method, and (iii) the positions of the arguments taking part in the calling pattern.

In the `setInterval` example above every subset of the parameters which includes either `left` or `right` is a proper calling pattern making a total of six calling patterns. We have therefore five auxiliary methods, whose mangled names and signatures are

1. `setInterval.int.int.int_0(int)`
2. `setInterval.int.int.int_2(int)`
3. `setInterval.int.int.int_0_1(int, int)`
4. `setInterval.int.int.int_1_2(int, int)`
5. `setInterval.int.int.int_0_2(int, int)`

Such names are possible since the JAVA virtual machine does not require method names to be valid JAVA identifiers. In fact, we insist on using invalid JAVA identifiers to avoid clashes with other user defined names.

The generation of the auxiliary methods is carried out in the parsing phase of the compiler so that these methods are being attributed and type checked as if they appeared in the source code.

The current implementation does not warn the user if a calling pattern of one method collides with a calling pattern of another method. (This situation happens only if the two methods have the same name, that is, in case of overloaded methods.) For example, the function call `Y.g(a :=3)` is ambiguous if class `Y` is defined as in Figure 4.3.1.

**Fig. 4.3.1 Two method definitions leading to an ambiguous calling pattern**

```java
class Y {
    static void g(int a, int b = a) {
        System.out.println("Two arguments");
    }
    static void g(int a) {
        System.out.println("One argument");
    }
}
```

95
The compiler does not warn against such a possibility while compiling class \( Y \), although it correctly refuses to make the ambiguous call. Other ambiguous situations may occur due to the combination of positional call and default parameters. In the above example, the call \( Y.g(17) \) is also ambiguous, and this ambiguity would not have been removed if the first argument of one of the functions was renamed.

The preferred order of evaluation of initialization expressions is left-to-right. That is, whenever two arguments can be computed based on the known values of other arguments, then the method realizing a calling pattern computes first the argument occurring first in the parameters list. In other words, the basic step executed repetitively by a method realizing a calling pattern is computing the leftmost computable argument. For example, consider method \( f \) defined by

```java
void f(int x = y+3, int y = 0; int z = 1){
    // ...
}
```

and its parameter-less invocation \( f(); \). The missing arguments are scanned from left to right. The first argument, \( x \), can not be computed until the value of the \( y \) parameter is obtained. Therefore, the \( y \) parameter is first computed. Next, the remaining missing arguments (that is, \( x \) and \( z \)) are scanned again from left to right. In this iteration parameter \( x \) can be computed based on the value of \( y \) as computed in the previous iteration. The order of evaluating the arguments is therefore \( y \) followed by \( x \) and finally \( z \).

**Default parameters and inheritance**

An overriding method might change the names of the formal parameters, but still in a method invocation the binding of parameters is done by their name in the static type, as depicted in Figure 4.3.2.

**Fig. 4.3.2** Overriding a method while changing a parameter name.

```java
class X {
    boolean equals(Object x) { /* ... */
        // ...
    }
}
class Y extends X{
    boolean equals(Object y) { /* ... */
        // ...
    }
}
```
In this figure method `equals` declared in class `X` is overridden in class `Y` while the name of the parameter is changed. With this hierarchy, the following invocation of `equals` is legal:

```java
X x1 = new Y();
x1.equals(x := new Y());
```
since the compiler looks for matching methods based on the static type of the receiver.

An overriding method may also change the initialization expression of a parameter provided that the set of legal calling patterns is not reduced, that is, the set of auxiliary methods of the overridden method is contained in the set of auxiliary methods of the overriding method.

For example, the following overriding of the method `setInterval` is legal:

```java
public void setInterval(
    int left = 0,
    int width = 0,
    int right = 0) { /* ... */
}
```

The set of auxiliary methods generated by the compiler for the overriding method contains that of the original method. However, a method declared as

```java
public void setInterval(
    int left, 
    int width = 0,
    int right = left+width) { /* ... */
}
```

is not a legal overriding of the same method `setInterval` since a calling pattern which omits `left` is a proper pattern for the original method but not for the overriding method. In this case the compiler issues an error regarding the incompatible default values.

Method Invocation

When it encounters a method invocation, a JAVA compiler first computes the set of all applicable methods and then looks for the most specific method among all applicable methods. We had to modify these steps in order to support methods with default arguments.
When checking the applicability of a method to a method invocation with keyword arguments, the actual parameters are reordered to match the order of the currently checked method. The usual applicability check is applied to the reordered parameters. For example consider the invocation $f(‘a’, s=”, x=1)$;

Fig. 4.3.3 Two overloaded method definitions.

$f(\text{char } t, \text{int } x, \text{String } s)\{\ldots\}$
$f(\text{char } u, \text{Object } s, \text{int } x)\{\ldots\}$

When this invocation is tested against the first method of Figure 4.3.3 the actual parameters are reordered so that ‘$a$’, which is an unnamed parameter, remains the first in the list, then 1, which corresponds to the second formal argument and finally ‘”, which corresponds to the formal argument named $s$. When the same invocation is tested against the second declaration in the figure, the order of the actual parameters is ‘$a$’ followed by ‘” and then 1.

Not only methods whose names are identical to the name of the invoked method are applicable. The applicability test is also applied to mangled methods generated from a method declaration whose name matches that of the invoked method.

Finding the most specific method is done by comparing pairs of methods. Each comparison eliminates the least specific one. Before the comparison our modified compiler sorts all formal parameters of the two methods for which the corresponding actual arguments in the invocation are named, by lexicographic order of their names and then applies the usual selection algorithm. For example, both methods declared in Figure 4.3.3 are applicable for the invocation above. Therefore, their arguments are reordered so that the first argument, which is unnamed in the invocation, remains as is, the second one is $s$ and finally $x$. Since these methods differ only in the second argument, and String is a subtype of Object, the first method is more specific than the second one.

4.3.2 Constructors with Default Arguments

Constructor Declaration

The JAVA virtual machine requires every constructor to have the special internal method name $<\text{init}>$. Consequently, the method for realizing the different auxiliary methods, that is, using mangling, can not be applied for constructors. Therefore, we encode the difference between auxiliary constructors in the types of the
formal parameters of these constructors rather than in their names. Every calling pattern of a constructor accepts, in addition to a subset of the original constructor arguments, a designated first argument whose type encodes the specific subset that this calling pattern represents. The auxiliary types are inner classes, residing in the constructor’s enclosing class, generated by the compiler during the parsing of a constructor with default values. For example, a constructor of an Interval class declared as:

```java
public Interval(int left = right - width,
               int width = 0,
               int right = left + width) {
    /* ... */
}
```

has, just like the setInterval method above, six calling patterns and five auxiliary constructors whose signatures are:

1. `Interval(JTC.int.int.int_0, int)`
2. `Interval(JTC.int.int.int_2, int)`
3. `Interval(JTC.int.int.int_0_1, int, int)`
4. `Interval(JTC.int.int.int_1_2, int, int)`
5. `Interval(JTC.int.int.int_0_2, int, int)`

As can be seen, the first parameter of each of the auxiliary constructors encodes both the types of the parameters in the original constructor and the positions of the parameters of the calling pattern. All the auxiliary types implement a common interface in order to distinguish auxiliary constructors from user defined ones, and their names start with a “JTC” prefix which stands for Java Type of Constructor.

The body of an auxiliary constructor, just like the body of an auxiliary method, is composed of missing parameters initialization followed by an invocation of another constructor. This pattern is not valid in the JAVA language, which dictates that a constructor invocation can only be the first statement in a constructor body however, this requirement is not enforced by the JVM, where a constructor invocation can follow other instructions.
Constructor Invocation

The process of choosing a constructor to invoke is similar to method invocation, that is, computing all applicable constructors and then selecting the most specific one. When computing applicable constructors, the modified compiler first examines each of the user defined constructors. This step is similar to computing applicable methods and includes reordering of the actual parameters. In order to examine the auxiliary constructors, a dummy null argument is prepended to the actual parameters to be matched against the distinguishing formal parameter.

Selecting the most specific constructor is similar to selecting the most specific method, however, if one of the compared constructors contains a distinguishing formal parameter this parameter is ignored in this stage.

4.4 Related Work

The case for- (and against-) keyword and default arguments was previously made in the literature. In this section, we review this historical scholarly discussion, and then briefly discuss the contemporary relevance of these, mostly historical arguments.

As early as 1976 Hardgrave [59] urged language designers to make the “additional effort to include keyword parameter technique in their languages” on the basis of the enhanced readability of this technique (with thoughtful selection of parameter names) and its flexibility in supporting changes to the order of parameters. The author, who was the first to argue for this language extension, also suggested a mechanism for supplying compile-time constant default values to parameters whose values are seldom “different from the standard”, and argued that these defaults should relieve clients from the chore of repetitively supplying values to these special arguments.

Three difficulties with default- and keyword- based invocation mechanisms were identified by Hardgrave: (i) unnecessary verbosity, which reaches an extreme in the case of single parameter- methods; (ii) compile time performance decrease, since the compiler has to bind each actual argument to a formal parameter; (iii) unintentional parameter omission may go undetected since the default values would be used.

In 1977, Francez [41] noticed that that keyword-based calling mechanism can be used to reflect not only the name of the formal parameter an argument is bound
to, but also the *kind* of binding, that is, “call by value” vs. “call by reference” vs. “call by name”. His work even included a concrete proposal for extending the keyword based notation to make this distinction. A year later, Parkin [104] drew the proponents’ attention to the fact that keyword based calls may lead to an undesirable confusion between the invoking- and the invoked- scope.

In 1982, Ford and Hansche [40] claimed that the keyword based method requires a “*somewhat cumbersome coding required in the actual parameter list*”. The authors also disliked the practice of writing, e.g., \( p(, a, , c , , c , , x , ,) \); to denote missing parameters with defaults in the positional method. Instead, they proposed extending the syntax already present for supplying formatting options to **PASCAL’s** built-in `write` procedure call, to user defined subprograms.

Two years later, Winkler [137] included keyword based calls to subprograms among his proposed additions and improvements to ISO-Pascal, while noting an interesting difficulty in passing subprograms as parameters to other subprograms. The mechanisms however did not make it into the language, nor to C or **OBERON** [139], languages which may be perceived by some as **PASCAL**’s successors.

Interestingly, **ADA** designers, observing [66] that the positional scheme suffers from the disadvantage that

“…with more than three or four parameters it is hard to follow the text.”

decided to include in their language a keyword based calling scheme arguing that it “*provides especially high readability*”. This calling scheme was used alongside with the “*almost universal*” positional calling mechanism. These four scholars also made the case for allowing both mechanisms in tandem arguing that

“Clearly in many contexts the order of parameters is either highly conventional (as for coordinate systems) or immaterial (as in \( \text{MAX}(X, Y) \)). Hence **ADA** admits both conventions. The classical positional notation may be used whenever the programmer feels that keyword parameters would add verbosity without any gain in readability.”

**ADA** thus allowed mixed positional- and keyword-based- calling mechanism, which was further enriched, as in our **JAVA** extension, with a default parameter facility which together provide “*a high degree of expressivity and readability*”.

The “selectors” of *keyword methods* of **SMALLTALK** and of **OBJECTIVEC** [29], e.g., the two arguments `at: put:` selector, resemble keyword-based- calling
scheme. The similarity is superficial since the invoker may not change the order of parameters, and \texttt{at: put:} is entirely different from the selector \texttt{put: at:}. Indeed, earlier versions of the language [74] did not even insist on using a keyword in front of every parameter, allowing invocations such as
\begin{verbatim}
displayFrame put ‘hi there’ at 150 100
\end{verbatim}

In the course of years, several languages in common use, including PERL [132], PYTHON and Transact-SQL stored procedures, have adopted a keyword-based calling scheme. Some of the historical arguments, for- and against- keyword based calling are clearly defunct now: the overhead in compilation time is negligible; Parkin’s comment on the confusion of scopes is not as relevant with the growing tendency of using small scopes; The syntax of Ford and Hansche failed to spike enthusiasm; and, the verbosity of keyword-based calls is addressed by the growing understanding that if keyword based parameter passing is allowed, it should be in addition to the positional scheme.

The arguments that stayed are that keyword-based calls are, when used appropriately, more flexible, more readable, and more expressive. Indeed, a recent internet page\textsuperscript{6} continues the debate, reiterates these arguments, also points out that despite disagreement whether it is easier to remember parameters based on their name and their position, such memorization is totally irrelevant with modern development environments. Similarly, such environments minimize the effort of maintaining the correctness of an invocation in the face of changes to the parameter names or to their order.

The third disadvantage pointed by Hardgrave, that is the risk of unintentionally omitting parameters, is still applicable. However, undetected semantic mistakes may occur in positional invocations as well, for instance, when switching two arguments of the same type in a method invocation.

This combination is used now in several programming languages including PYTHON, LISP and MESA as well as C# 4.0. However, to the best of our knowledge, there was no study of the confusion and abuse that this combination may create.

\textsuperscript{6}http://c2.com/cgi/wiki?KeywordParameterPassing
4.5 Conclusions

In this chapter we argued that there is a clear and present need for an inherent support of keyword and default arguments in JAVA. We based our claim on previously published work, in which it was determined that the most frequent use of overloading is for simulating defaults arguments; this use pattern occurring in about a third of the cases in which overloading is used.

We discussed the advantages of a designated keyword and defaults mechanism in JAVA over the existing solution of method overloading in terms of code length, implications on documentation, maintenance and client’s learning curve, and flexibility in handling situations in which several arguments are of the same type.

We bring a proof of concept implementation of keyword and default arguments in JAVA. Our implementation, which does not impinge on the runtime environment, allows supplying defaults to any subset of the formal parameters, and admits method invocations in a *positional* style, *nominal* style (i.e., where the role of parameters is determined by prefixing these with their name rather than their position in the arguments list), or *mixed* style. We argue that this extension may address the famous options-operands dilemma well.

Although such a mechanism can drastically reduce the amount of overloading in JAVA code, it raises its own questions of abuse and confusion. For example, the interaction of default arguments and overloading may lead to confusing semantics and ambiguity. Figure 4.3.1 depicts a situation in which an ambiguous method call may occur. The solution we chose in our extension is to report an error for the ambiguous call. C#’s implementation is different, giving precedence to methods that have no default arguments. In both cases the interaction of default arguments and overloading makes the code confusing and hard to maintain.

There are other choices made in our extension which are different from C#. For example, the evaluation order of the arguments. Our implementation scans the arguments by the order in which they appear in the method *definition*, while arguments are evaluated in their order in the method *invocation* in C#. This difference is significant if evaluating the arguments has side effects.

The implementation of named and optional arguments in C# differs from our extension also by the restrictions imposed on initialization expressions. While in C# only constants may be used as default values, our extension is more flexible, allowing method invocations, data members access and other formal arguments to be used in initialization expressions.
Our design choice of extending the language without modifying the JVM requires the creation of synthetic methods and types. While this implementation approach is sometimes used in Java compilers (e.g., enums introduce additional types, bridge methods are generated by the compiler to support return type covariants), it may effect runtime tools such as profilers and debuggers, and confuse programmers that use reflection.

The effect of the language extension on performance should be measured. We expect that the extension carries a performance penalty during compilation, mainly due to parameters reordering done for method binding. But, of course a comprehensive benchmark is required to measure the performance effect both on compilation time and on execution time.
Chapter 5

A Microbenchmark Case Study and Lessons Learned

Science is all about the generation of new and verifiable truths. In experimental sciences, this amounts to reproducible experimentation. In experimental computer science, reproducibility may seem easy, since we are accustomed to deterministic and predictable computing systems: We expect that hitting a key labeled ‘a’ shall always, in a bug-free environment, produce the letter ‘a’ on the screen.

It came therefore as a surprise to us that in our attempt to benchmark a certain small JAVA function, a task known as micro-benchmarking, we encountered inconsistent results, even after neutralizing effects of well known perturbing factors such as garbage collection, just in time compilation, dynamic loading, and operating system background processes.

This chapter tells the story of our micro-benchmarking endeavors, and tries to characterize the inconsistencies we encountered. Little effort is spent on trying to explain these; we believe this task requires a dedicate study which would employ different research tools (simulation and hardware probes come to mind). Our hope is that this report would contribute to better understanding of the phenomena we describe and promote the development of methods to eliminate these.

Chapter Outline. The remainder of this chapter is organized as follows. Section 5.1 discusses some of the hardships of micro-benchmarking of JAVA code. In Section 5.2 we outline the main findings of our work. Section 5.3 then surveys related work in the area of non-determinism of benchmarking result. In Section 5.4 we describe the setting of our experimental evaluation, the hardware and software
used, and the benching environment. Section 5.5 shows that different, seemingly identical, invocations of the JVM may converge to distinct steady states. This finding is further examined in Section 5.6, which also shows that the discrepancy is greater when the JIT compiler is not present. Section 5.7 shows that even a single JVM invocation may have multiple steady states, and that, further, these multiple steady states may be simultaneous. The impact of prologue execution is the subject of Section 5.8. Section 5.9 concludes and suggests directions for further research.

5.1 Background: Benchmarking and Micro-benchmarking of Java

Benchmarking of computer systems is a notoriously delicate task [61]. The increasingly growing abstraction gap between programs and execution environments, most notably virtual machines, forces benchmarking, which could, in the early days, be carried out by counting program instructions, to use methods used in experimental, exact and social sciences.

Other issues brought about by the widening abstraction layer include the fact that the same benchmark on different platforms may yield different, and even contradictory results [14]. In addition, different compilers may apply different optimizations to source code, and therefore comparing two alternatives compiled with one compiler can lead to different results than the comparison of the same alternatives compiled using a different compiler.

Micro-benchmarking, measuring the performance of a piece of code (as opposed to assessing the performance of an application) is even more challenging. There are several subtle issues that can cause a benchmarker to draw wrong conclusions from an experiment. One such aspect is the use of dead code: microbenchmarks often do nothing but calling the benchmarked code. Compilers may recognize such pattern and eliminate parts of the benchmark code, leading to code that runs faster than expected. In order to avoid that, it is not enough just to call the benchmarked code, but some extra code has to be introduced in the benchmark. However, doing so leads to a benchmark which measures the performance of the original and the extra code, introducing noise into the measurement.

Micro-benchmarking of JAVA functions raises its own set of intriguing issues.1

---

1See Clikc’s JavaOne’02 presentation, “How NOT To Write a Microbenchmark?”,
There are two major hurdles for Java microbenchmarks: First, the Just In Time (JIT) compiler, may change the code as it executes. Secondly, the Garbage Collector (GC) may asynchronously consume CPU cycles during the benchmark. Multithreading may also be a concern, given the fact that an ordinary Java Virtual Machine (JVM) invocation, even of a non-threaded program, spawns a dozen of threads in addition to the main thread.

JIT compiles methods to different optimization levels, and is driven by timer-based sampling. Thus, in different runs different samples may be taken, leading to different methods being optimized at different levels, and to variations in running time of a benchmark across different JVM invocations, even on a single host [4].

The first iterations of the benchmarked code include a large amount of dynamic compilation. Later iterations are usually faster both since they include less compilation and because the executed code is compiled and optimized [43]. The common wisdom of dealing with the presence of the JIT compiler is by conducting, prior to the benchmark, warm-up executions of the benched code. The warm-up stage should be sufficiently long to allow the JIT compiler to fully compile and optimize the benchmarked code.

In dealing with the GC, one can allocate sufficiently large heap space to reduce the likelihood of GC. Also, the intermittent nature of the GC effect on the results can be averaged by repeating the benchmark sufficiently many times.

The main tool of the trade is then replication: “billions and billions of runs” is the phrase sometimes used by practitioners. But, as we discovered, there are certain contributions to inconsistency which could not be mitigated by simple replication.

5.2 Findings

Having eliminated as much as we could the effects of JIT compilation, GC cycles and operating system and other environmental noise, we conducted benchmarking experiments consisting of a very large number of runs. Our case study revealed at least four different factors that contributed to inconsistency in the results:

1. Instability of the Virtual Machine. We show that different, seemingly identical invocations of the JVM may lead to different results, and that this difference (which may be in the order of 3%) is statistically significant. The impact of this instability may be more significant when two competing implementations are compared.
This variation increases with the abstraction level, in the sense that when the JIT compiler is disabled, the discrepancy between different VM invocations increases.

Observe that naive replication, within the same VM cannot improve the accuracy of the results. The remedy is in replication of the VMs.

2. *Multiple Steady States.* We observed that in some invocations, the VM converges to a certain steady state, and then, within the same invocation, leaps into a different steady state. The consequence of this phenomenon is that an averaged measurement within a single VM invocation may be misleading in not reflecting or characterizing these leaps.

3. *Multiple Simultaneous Steady States.* Further, we observed that in some invocations, the VM converges to two or more simultaneous steady states, in the sense that a given measurement has high probability of assuming one of number of distinct results, and small probability of assuming any intermediate value. A simple average of these results may be even more misleading report of the benchmarking process.

4. *Prologue Effects.* Finally, we demonstrate that the running-time of the benchmarked code may be significantly affected by execution of unrelated prologue code, and that this effect persists even if the benchmarked code is executed a great number of times. This means that one cannot reliably benchmark two distinct pieces of code in a single program execution.

The more disturbing conclusion is that the timing results obtained in a clean benchmarking environment may be meaningless when the benchmarked code is used in an application. Even if the application executes this code numerous times, the (unknown) application prologue effects may persist, rendering the benchmarked results meaningless.

### 5.3 Related Work

Non-determinism of benchmarking results is a well studied area. It is well known that modern processors are chaotic in the mathematical sense, and therefore analyzing the performance behavior of a complex program on a modern processor architecture is a difficult task [12]. Eechout et al. [36] showed that benchmarking result highly depends on the virtual machine; results obtained for one VM may
not be obtained by another VM. Blackburn et al [14] showed that a JAVA performance evaluation methodology should consider multiple heap sizes and multiple hardware platforms, in addition to considering multiple JVMs.

Several attempts were made to suggest methodologies for benchmarking. One particularly interesting and increasingly widely used methodology, replay compilation [65], used to control the non-determinism that stems from compilation and optimization. The main idea is the creation of a “compilation plan”, based on a series of training runs and then the use of this plan to deterministically apply optimizations.

Georges, Buytaert and Eeckhout [43] describe prevalent performance analysis methodologies and point their statistical pitfalls. The paper presents a statistically robust method for measuring startup time and steady-state performance.

Blackburn et al. [15] recommend methodologies for performance evaluation in the presence of non-determinism introduced by dynamic optimization and GC in managed languages. Their work stresses the importance of choosing a meaningful baseline, to which the benchmark results are compared, and controlling free variables such as hosts, runtime environments, heap size and warm-up time. To deal with non-determinism the authors suggest three strategies: (i) using replay compilation, (ii) measuring performance in steady state, with JIT compiler turned off, and (iii) generating multiple results and apply statistical analysis to these.

Mytkowicz, Diwan, Hauswirth and Sweeney [98] discuss factors which cause measurement bias, suggest statistical methods drawn from natural and social sciences to detect and avoid it. To avoid bias, the paper suggests a method which is based on applying a large number of randomized experimental setup, and using statistical methods to analyze the results. The method for detecting bias, causal analysis, is a technique used for establishing confidence that the conclusions drawn from the collected data are valid.

Lea, Bacon and Grove [78] claim that there is a need to establish a systematic methodology for measuring performance, and to teach students and researchers this methodology.

Georges et al. [44] present a technique for measuring processor-level information gathered through performance counters and linking that information to specific methods in a JAVA program. They argue that “different methods are likely to result in dissimilar behavior and different invocations of the same method are likely to result in similar behavior”.

109
5.4 Settings of the Benchmark

5.4.1 Hardware and Software Environment

To minimize effects of disk swapping and background processes on benchmarking, we selected a computer with a large RAM and multiple cores. Measurements were conducted on a Lenovo ThinkCentre desktop computer, whose configuration is as follows: an Intel Core 2 Quad CPU Q9400 processor (i.e., a total of four cores), running at 2.66 GHz and equipped with 8GB of RAM. Hosts used in specific experiments are described below.

During benchmarking all cores were placed in “performance” mode, thereby disabling clock rate and voltage changes.

On this computer, we installed Ubuntu version 10.04.2 (LTS) and the readily available open JAVA development kit (OpenJDK, IcedTea6 1.9.8) including version 1.6.0_20 of the JAVA Virtual Machine (specifically 6b20-1.9.8-0ubuntu1-10.04.1). Other software environments used in specific experiments are described below.

To minimize interruptions and bias due to operating system background (and foreground) processes, all measurements took place while the machine was running in what’s called “single-user” mode (telinit 1), under root privileges in textual teletype interaction (no GUI), and with networking disabled.

5.4.2 Benchmark Algorithm

Our atomic unit of benchmarked code was defined as an instance of a class implementing interface Callable (defined in package java.util.concurrent). This interface defines a single no-arguments method, which returns an Object value.

Given an implementation of Callable our benchmark algorithm works in three stages: (i) estimation, (ii) warm-up, and (iii) measurement. The purpose of the estimation phase is to obtain a very rough estimate of execution time of the parameter. In the warm-up stage, this estimation is used for warming up the given Callable, i.e., iterating it until the JIT compiler has realized its full potential, for a specified warm-up period (five seconds in our experiments). In the measurement phase, the running time of the parameter is measured repeatedly. Each such measurement executes the parameter \( m \) times, where \( m \) is so selected that the measurement duration is roughly that of a parameter \( \tau \).

All phases are based on making a number of sequences of iterations of the
Callable object parameter. Each iteration sequence is monitored for garbage collection cycles, JIT compilation cycles, and class loading/unloading events by probing the MX-beans found in class `ManagementFactory` (which is defined in package `java.lang.management`). The timing result of a sequence is discarded if any of these events happen. Further, the detection of JIT compilation cycles late in the warm-up period, increases its duration. And, the repeated detection of garbage collection cycles triggers shortening of a measurement sequence so as to decrease the likelihood of further interruptions of this sort.

5.4.3 Benchmarked Code

At the focus of our attention we placed the implementation of class `HashMap`, arguably one of the most popular classes of the JAVA runtime environment. We used version 1.73 of the code, (dated March 13, 2007) due to Doug Lea, Josh Bloch, Arthur van Hoff and Neal Gafter. To preclude the option of pre-optimization of code present in JAVA libraries, we benchmarked a copy of the code in our benchmarking library.

Within this class, we concentrated in the throughput of one function named `V get(Object key)`, which retrieves the value associated with a key stored in the hash table. Listing 5.4.1 depicts this function’s code.

This function includes iterations, conditionals and logical operations, array access and pointer dereferencing. Examining function `hash` which is called by `get` in line 3 (Listing 5.4.2), we see that the benchmarked code includes also bit operations.

```java
static int hash(int h) {
    h ^= h >>> 20 ^ h >>> 12;
    return h ^ h >>> 7 ^ h >>> 4;
}
```

**List 5.4.2:** JAVA code for function `hash` (invoked from `get` in class `HashMap`)

Our measurements never pass a `null` argument, so there is no need to examine the code of function `getForNullKey`. Also, we only exercise `get` in searching for keys which are *not present* in the hash table. The call `key.equals(k)` is thus executed only in the case that the cached hash value (field `hash` in the entry) happens to be equal to the hash value of the searched key. This rare event does not happen in our experiments, so, the call `key.equals(k)` is never executed.
public V get(Object key) {
    if (key == null) return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e=table[hash & table.length-1]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash &&
            (k = e.key) == key ||
            key.equals(k))
            return e.value;
    }
    return null;
}

List 5.4.1: Java code for benchmarked function (function get in class HashMap)

All in all, the benchmarked code is presented in its entirety in listings 5.4.1 and 5.4.2. Notably, the benchmarked code does not include any virtual function calls—foregoing virtual optimization techniques. Viable optimization techniques include, e.g., branch prediction, pre-fetching, inlining, and elimination of main memory access operations. Also, the code does not allocate memory, nor does it change the value of any reference. As it turns out, there was not a single garbage collection cycle in a sequence of one billion consecutive calls to function get.

The precise manner in which get was exercised was as follows:

1. Fix \( n = 1,152 \).

2. Make \( n \) pairs of type \((\text{key}, \text{value})\), where both the String key and the Double value are selected using a random number generator started with a fixed seed.

3. Create an empty HashMap<String, Double> data structure and populate it with these pairs.
4. Create an array of \( n \) fresh keys of type \texttt{String}, where keys are selected using the same random number generator.

5. The benchmarked procedure, \texttt{getCaller()}, then iterates over the keys array, calling \texttt{get} for each of its elements.

6. The \textit{throughput} of function \texttt{get} is defined as the running-time of procedure \texttt{getCaller()} divided by \( n \).

### 5.5 No Single Steady State

We conducted \( v = 7 \) independent \textit{invocations} of our benchmarking program, i.e., each invocation uses a freshly created VM. These invocations were consecutive, with a script invoking the VM \( v \) times.

In each invocation the VM carried out \( r = 20,000 \) \textit{iterations}, i.e., \( r \) executions of the measurement procedure described above (after a single estimation stage followed by a single warm-up stage). We shall call such an iteration a \textit{measurement session}. A measurement session returns a \textit{measurement result} (result for short).

A measured result of \( t \) (for function \texttt{getCaller()}) gives a throughput of \( t/n \) for function \texttt{get}(). Also, a measurement session of length \( \tau \) calls \texttt{getCaller()} about \( \tau/t \) times. The throughput in our experiment was about 15.3\text{ns} per each \texttt{get}() call.

Overall, in our case, \( \tau/t \approx 6,000 \). If the time of each of the iterations of \texttt{getCaller()} were independent, then, by the \textit{central limit theorem} the \( r \) results collected in an invocation should be close to a normal distribution.

Overall, in the experiment described in this section, the number of executions of function \texttt{get}, was about

\[
v \times r \times \frac{100\text{ms}}{15.3\text{ns} \times 1,152} \approx 0.8 \times 10^9
\]

Table 5.1 summarizes the essential statistics of the distribution of measurement results in our experiment. (The “mad” column represents the median of absolute deviations from the median—a statistics which is indicative of the quality of the median statistics and robust to outliers.)

The values in each column, i.e., the same statistics of different invocations, appear quite similar. The median is close to the mean; these statistics indicate a throughput of about 15.2–15.3\text{ns}/operation. The standard deviation is about 0.1\text{ns},
<table>
<thead>
<tr>
<th>No.</th>
<th>Mean</th>
<th>$\sigma$</th>
<th>Median</th>
<th>Mad</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.27</td>
<td>0.0802</td>
<td>15.27</td>
<td>0.0509</td>
<td>14.95</td>
<td>15.69</td>
</tr>
<tr>
<td>2</td>
<td>15.17</td>
<td>0.0768</td>
<td>15.17</td>
<td>0.0454</td>
<td>14.74</td>
<td>15.69</td>
</tr>
<tr>
<td>3</td>
<td>15.24</td>
<td>0.0726</td>
<td>15.24</td>
<td>0.0451</td>
<td>14.93</td>
<td>15.67</td>
</tr>
<tr>
<td>4</td>
<td>15.38</td>
<td>0.0727</td>
<td>15.38</td>
<td>0.0430</td>
<td>15.06</td>
<td>15.81</td>
</tr>
<tr>
<td>5</td>
<td>15.29</td>
<td>0.0867</td>
<td>15.30</td>
<td>0.0495</td>
<td>14.95</td>
<td>15.68</td>
</tr>
<tr>
<td>6</td>
<td>15.10</td>
<td>0.0722</td>
<td>15.11</td>
<td>0.0420</td>
<td>14.83</td>
<td>15.51</td>
</tr>
<tr>
<td>7</td>
<td>15.24</td>
<td>0.0929</td>
<td>15.25</td>
<td>0.0559</td>
<td>14.87</td>
<td>15.60</td>
</tr>
</tbody>
</table>

which is about 0.7% of the mean. The extreme values appear to be well behaved in that they deviate from the mean by about 5%.

The mean throughput in the different invocations spans a $0.28\text{ns} \approx 2\%$ range which may seem reasonable. A graphical plot of the distribution of the same results, as done in Figure 5.5.1, reveals a somewhat different picture. A quick look at the figure, suggests that the results in two different invocations are drawn from close, yet distinct, distributions.

Fig. 5.5.1 Distribution of benchmark results in seven invocations of a 20,000 iterations experiment, $\tau = 100\text{ms}$

Each curve in the figure represents the distribution function of the $r$ values obtained in a single invocation. Here, and henceforth, all curves were smoothed out by using a *kernel density estimation*, where a Gaussian was used as the windowing function, and where a bandwidth free parameter was $h$ (lower values means less...
estimation). The value of \( h \) selected in all figures is

\[
h = \min_i \frac{4}{3r} \frac{1}{\sigma_i},
\]

where \( i \) ranges over all distribution plotted in the same figure, \( \sigma_i \) being the standard deviation of the \( i^{th} \) distribution. (In normal, Gaussian distribution, the optimal value of \( h \) is \((4/3r)^{1/5}\sigma_i\)).

Also, here and henceforth, the smoothed distribution was scaled down by multiplying the density by \( 1/r \), so as to make the total area under the curve 1.

A quick “back of an envelope” analysis confirms that the difference between the means of the different invocations is statistically significant.

To increase certainty in our findings, we repeated the experiment with \( \tau = 1,000\text{ms} = 1\text{sec} \), except that for practical reasons, we set \( v = 5 \). The resulting distributions are depicted in Figure 5.5.2.

**Fig. 5.5.2** Distribution of benchmark results in five invocations of a 20,000 iterations experiment, \( \tau = 1,000\text{ms} \)

To appreciate the scale, note that each invocation this time executed function \( \text{get} \approx 1.1 \text{ billion times. The actual benchmarked function, i.e., the function which wraps get, calling it } n = 1,152 \text{ times, is called about 60,000 times in each of the 20,000 iterations.} \)

The overall picture portrayed by Figure 5.5.1 and Figure 5.5.2 is that invocations of the JVM, even in our fairly clean benchmarking does not converge to a single steady state. It seems as if the mythical single convergence point is replaced by multiple steady states, or even a range of convergence centers. The figures suggest that in this particular case, this range spans about 2% of the measured value.
The consequence is that repeated invocations of a single benchmark shall yield different results, even if all factors are kept equal. Further, when one compares competing algorithms or implementations, this error accumulates and may bring about wrong conclusions.

It is interesting to see that each of the distributions plotted in Figure 5.5.1 resembles normal distribution. Figure 5.5.3 compares one such distribution with that of a normal distribution with the same mean and standard deviation.

Fig. 5.5.3 Distribution of benchmark results in an invocation of a 20,000 iterations experiment, \( \tau = 100 \text{ms} \), compared to a Gaussian with the same mean and standard deviation.

Evidently, the experimental distribution is even more peaked than a Gaussian. Applying a Jarque-Bera test we obtained that it very highly unlikely \( p < 10^{-5} \) that the measurement sessions’ results were drawn from a normal distribution. This fact is an initial indication that the measurement sessions’ results are not independent.

Still, we may conservatively assume that the distribution of the results is normal, for, e.g., computing the standard error. This is useful for estimating accuracy of the mean, even for small number of iterations. In selecting, for example, \( r = 10 \), the standard deviation would be about \( 0.1\text{ns} \times \sqrt{10/9} \), i.e., a standard error \( \approx 0.06\text{ns} \) which gives about 0.2% measurement error. In selecting a greater number of iterations the measurement error could be further decreased, i.e., for \( r = 100 \), executing in about ten seconds, the measurement error would be about 0.1%.

The difficulty in applying this analysis for benchmarking is that different invocations lead to slightly different distributions centered at different values. The inherent 2% discrepancy of these cannot be improved by increasing \( r \). Further,
as we shall see below, even a single invocation of the JVM may have multiple convergence points.

5.6 Examining the No Single Steady State Judgment

One may hypothesize that the discrepancy of the results can be explained by the fact that the invocations were consecutive. The CPU, hardware cache and other factors may have a long “learning” curve. These can, presumably, act together so that the computing carried out in one experiment somehow improves the results obtained in the subsequent experiment, despite execution in a different process of the operating system and in a distinct address space.

A quick look at Table 5.1 indicates that this could not be the case, since later invocations do not yield lower values than those of earlier invocations. The respective statistics of the longer experiment (Figure 5.5.2), provided in Table 5.2 do not portray a different picture.

<table>
<thead>
<tr>
<th>No.</th>
<th>Mean</th>
<th>σ</th>
<th>Median</th>
<th>Mad</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.17</td>
<td>0.0585</td>
<td>15.17</td>
<td>0.0392</td>
<td>14.95</td>
<td>15.40</td>
</tr>
<tr>
<td>2</td>
<td>15.21</td>
<td>0.0859</td>
<td>15.21</td>
<td>0.0483</td>
<td>14.88</td>
<td>15.73</td>
</tr>
<tr>
<td>3</td>
<td>15.23</td>
<td>0.0828</td>
<td>15.23</td>
<td>0.0510</td>
<td>14.90</td>
<td>15.68</td>
</tr>
<tr>
<td>4</td>
<td>15.05</td>
<td>0.0687</td>
<td>15.04</td>
<td>0.0391</td>
<td>14.75</td>
<td>15.57</td>
</tr>
<tr>
<td>5</td>
<td>15.21</td>
<td>0.0756</td>
<td>15.20</td>
<td>0.0429</td>
<td>14.92</td>
<td>15.69</td>
</tr>
</tbody>
</table>

We see in Table 5.2 that the centers of the distributions are the same as in Table 5.1. The standard deviation and other statistics are also about the same. And, Table 5.2 does not suggest that a certain invocation is favorably affected by earlier invocations.

Is the absence of the single steady state an artifact of instability of the triggering of optimization algorithms within the JIT? To answer this question, we repeated the experiment while disabling the JIT compiler (by passing `-Djava.compiler=NONE` flag to the JVM). The resulting distributions are depicted in Figure 5.6.1.

Comparing the figure with Figure 5.5.1 and Figure 5.5.2, we see that in the absence of the JIT compiler the distributions and their steady states are even more
Fig. 5.6.1 Smoothed distribution function of benchmark results, JIT compiler disabled, in five invocations of a 20,000 iterations measurement, $\tau = 100$ms

![Smoothed distribution function of benchmark results, JIT compiler disabled, in five invocations of a 20,000 iterations measurement, $\tau = 100$ms](image)

clearly separated. Examining Table 5.3, which summarizes the respective statistics of these distributions, strengthens our conclusion.

**Tab. 5.3** Essential statistics (in ns/operation) of the distribution of measurement results, JIT compiler disabled, in five consecutive invocations of a 20,000 iterations measurement, $\tau = 100$ms.

<table>
<thead>
<tr>
<th>No.</th>
<th>Mean</th>
<th>$\sigma$</th>
<th>Median</th>
<th>Mad</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>231.04</td>
<td>0.1642</td>
<td>231.07</td>
<td>0.0888</td>
<td>229.69</td>
<td>231.76</td>
</tr>
<tr>
<td>2</td>
<td>212.00</td>
<td>0.0980</td>
<td>211.99</td>
<td>0.0651</td>
<td>211.71</td>
<td>212.44</td>
</tr>
<tr>
<td>3</td>
<td>212.76</td>
<td>0.0819</td>
<td>212.75</td>
<td>0.0508</td>
<td>211.89</td>
<td>213.20</td>
</tr>
<tr>
<td>4</td>
<td>229.68</td>
<td>0.1868</td>
<td>229.72</td>
<td>0.0747</td>
<td>227.42</td>
<td>231.19</td>
</tr>
<tr>
<td>5</td>
<td>218.80</td>
<td>0.0921</td>
<td>218.79</td>
<td>0.0554</td>
<td>218.12</td>
<td>219.74</td>
</tr>
</tbody>
</table>

As can be seen in the table, there is a 9% range span between the best result, in which the average throughput was 212ns/operation, and the worst result of 231.04ns/operation. This difference is even more significant considering the fact that the coefficient-of-variance (CoV), defined as the standard deviation divided by the mean, is smaller in the absence of the JIT. The greatest CoV in Table 5.3 is 0.07%, while the smallest CoV in tables 5.1 and 5.2 is 0.39%.

As argued by Blackburn et al. [14], the choice of architecture may affect the reported results substantially. Therefore, we repeated the experiments on six additional different computing systems. Table 5.4 summarizes the hardware and software characteristics of the additional systems on which the experiments were conducted.
Note that all systems were operating under the Gnu/Linux operating system, and were all equipped with 2GB or more of internal memory. The main variety is the range of CPU models, along with their respective cache levels.

We carried out $v = 7$ VM invocations on each of systems A—F, where each invocation includes 20,000 iterations of measurements spanning $\tau = 100$ ms each. The distribution density functions are plotted in Figure 5.6.2.

On the least modern system A, we see almost identical, overlapping distributions. The mean of these distributions are identical up to three significant digits (39.4 ns/op, while the standard deviation is 0.2% of this value). The fact that similar results for different invocations were obtained in this system indicates that the multiple steady-state phenomenon is not due to an inherent problem in our benchmarking system, but a behavior which is observed only in certain environments.

In the slightly more modern system B, the distribution are further apart, with the means varying between 36.6 ns/op and 37.2 ns/op (a 1.6% range), while the standard deviations being about 1% of the mean.

We clearly have distinct steady states in systems C, D, and E. In system D, there are two distinct multiple steady states, one at 15.2 ns and another at 14.95 ns. These are about 1.5% apart. In system E there appear to be three distinct steady states which are 1.7% apart. System F exhibits the most interesting behavior: we see on it that there are multiple local maxima of the distribution density function.
Overall, the curves in Figure 5.6.2 lead us to conclude that the phenomena of absence of a single steady state for a different VM invocations on the same platform is not unique to our specific benchmarking environment, and that it is typical of more modern architectures.

5.7 Shift of Steady State

Having examined the host system, it is only natural to wonder whether the benchmarked code, i.e., function get may contribute its share to the indeterminism we encountered. To this end, benchmarking was repeated for four other functions written specifically for this study:

1. arrayBubbleSort(), which randomly shuffles the values in a fixed array of ints of size \( \ell_1 \), and then resorts these using a simple bubble sort algo-
2. `listBubbleSort()`, which randomly shuffles the values in a fixed linked list of `Integer`s of size \( \ell_2 \) (without changing the list itself), and then sorts these (again, without modifying the list) using the same bubble sort algorithm.

3. `xorRandoms()`, which computes the X-OR of \( \ell_3 \) consecutive pseudo random integers drawn from JAVA’s standard library random number generator.

4. `recursiveErgodic()`, which applies many recursive calls involving the creation and destruction of lists of integers, which are implemented using `ArrayList<Integer>` to create an array of pseudo-ergodic sequence of length \( \ell_4 \).

The implementation of these made the results of the computation externally available so as to prevent an optimizing compiler from optimizing the loops out. Note that each of the four functions has its own particular execution profile: Function `arrayBubbleSort` is characterized best by array access operations, while pointer dereferencing is the principal characterization of `listBubbleSort`. Function `xorRandoms()` involves a lot of integer operations and is therefore mostly CPU bound, while memory allocation is the main feature of `recursiveErgodic()`.

The values \( \ell_1, \ell_2, \ell_3, \) and \( \ell_4 \) were so selected to make the running-time of each of these four functions about the same as the function that wraps `get()`. Thus, for \( \tau = 100 \text{ms} \) we have again about 6,000 calls of the benchmarked function, and, we expect again of the sessions’ results to be normally distributed. Further, to make it easier for humans to compare the results, we selected integers \( n_1, n_2, n_3, \) and \( n_4 \) and divided the resulting running-time values by these numbers so as to make the “throughput” values close to those of the throughput of `get()`.

We conducted \( v = 3 \) VM invocations of each of these functions. As usual, each such invocation constituted \( r = 20,000 \) iterations of measurement sessions of \( \tau = 100 \text{ms} \). Each measurement session thus included about 60,000 executions of the benchmarked function. The distribution density curves for this experiment are plotted in Figure 5.7.1.

Benchmarking function `recursiveErgodic()` (Figure 5.7.1(d)) gives similar results to what we have seen before: Gaussian like distribution in each invocation, but different means in different invocations, i.e., the lack of single steady state.
Fig. 5.7.1 Distribution density of throughput of four auxiliary JAVA functions, $v = 3$, $r = 20,000$, $\tau = 100$ms.

(a) Function `arrayBubbleSort()`

(b) Function `listBubbleSort()`

(c) Function `xorRandoms()`

(d) Function `recursiveErgodic()`
Most interesting is Figure 5.7.1(c), depicting the distribution density of the measurements of function \texttt{xorRandoms()}. The three different curves of the three different invocations are so close to each other that they cannot be distinguished. This similarity is encouraging, since it tells us that the variations between the different invocations that encountered are not an artifact of the benchmarking environment. Moreover, we can conclude that the bell-shape distribution of results is not due to the usual fluctuations in measurement errors, but is inherent to the performance of the benchmarked function. Said differently, the time performance of (some) benchmarked functions obeys a Gaussian like distribution, whose variation is significantly greater than measurement errors introduced by the measurement environment. Also interesting is the fact that the distribution density has two very sharp peaks, one at the level of 15ns/op throughput and the other at 15.2ns/op. Similar, multi-peaked distribution occurs also in \texttt{arrayBubbleSort}. The peaks are not located at identical locations in different executions but they are even sharper (observe that the different y-axis scales).

This multi-peak phenomenon repeats itself also in Figure 5.7.1(b), so it could not be a coincidence. To understand this phenomenon better, we plotted the measurement result vs. session number for each of the four functions (Figure 5.7.2). To conserve space, the figure is restricted to the first invocation of each function.

Figure 5.7.2(d) shows that the results obtained in the \(r\) measurement sessions of function \texttt{recursiveErgodic()} are distributed more or less as expected, with the center of fluctuations not changing throughout the invocation.

Functions \texttt{arrayBubbleSort()}, \texttt{listBubbleSort()} and \texttt{xorRandoms()} are different in that the center of fluctuation, or “steady state” so to speak, changes during the run; these changes are always “step wise” rather than gradual.

The multiple steady states observed in Figure 5.7.1 correspond to these “quantum” leap of the center of fluctuations. Examining Figure 5.7.1(c) together with Figure 5.7.2(c) suggests that the quantum leap in function \texttt{xorRandoms} always occurs at about the same measurement session during the invocation.

In contrast, the local maxima of the distribution functions the different invocations of \texttt{arrayBubbleSort} and \texttt{listBubbleSort} do not coincide. This fact is an indication that the leaps of center of fluctuation are not necessarily deterministic. To appreciate this point better, consider Figure 5.7.3, in which the measurement session result is plotted against the session number for the second invocation of \texttt{listBubbleSort}. 

123
Fig. 5.7.2 Measurement result (throughput ns/ops) vs. session No. of the four auxiliary benchmarked functions.

(a) Function arrayBubbleSort

(b) Function listBubbleSort

(c) Function xorRandoms

(d) Function recursiveErgodic
Fig. 5.7.3 Measurement result (throughput ns/ops) vs. session No. of the second invocation of benchmark of function \texttt{listBubbleSort}.

Now, the comparison of this figure with Figure 5.7.2(b), depicting the first invocation of function \texttt{listBubbleSort()}, clearly indicates that the leaps between the different steady states do not occur at the same point during the invocation.

Recall that in the previous section, multiple local maxima were observed also in the distribution of results of benchmarking function \texttt{getCaller()}. It is therefore interesting to examine the progression of these results with time, as depicted in Figure 5.7.4.
Fig. 5.7.4 Measurement result (throughput ns/ops) vs. session No. of the four invocations of benchmark of function getCaller on system F (see Table 5.4 and Figure 5.6.2 above), $r = 20,000$, $\tau = 100\text{ms}$. 
As in Figure 5.7.2, we see that (i) the execution time of the benchmarked function `getCaller()` on System F fluctuates about multiple steady states; (ii) these steady states change in time; and, (iii) the leaps between steady states occur in a seemingly non-deterministic fashion. However, unlike what we have seen before, Figure 5.7.4 shows that multiple steady states may occur simultaneously, e.g., there are two equally weighted steady states during most of the invocations depicted in Figure 5.7.4.

5.8 The Effect of Prologue Runs

One of the frustrating issues in our attempts of benchmarking function `get()` was our suspicion that the results seemed to have changed a bit when minor modifications were applied to portions of the enveloping code, e.g., code in charge of command line arguments processing, measurement, bookkeeping and tuneup. To eliminate this yet unverified suspicion, all our experiments were carried out within a fixed version of the enveloping code.

Further, to check whether code executed prior to benchmarking may affect the results, we carried out an experiment in which actual benchmarking commenced after a prologue session in which \( m \) non-benching calls were made to foreign code. To make this foreign code similar to the benchmarked code, the calls in the prologues were made to an alternative function `getPrimeCaller`, which in turn made \( n = 1,152 \) calls to an alternative function named `getPrime()`. Function `getPrime()` does a similar, failing search on an alternative implementation of class `HashMap`, and its execution time is similar to that of `get()`.

After these \( m \) prologue calls, benchmarking of function `getCaller()` carried out as usual. The value of \( m \) was deliberately kept small: We maintained \( m \leq 100 \). Recall that in a single benchmarking session whose length is \( \tau = 100 \)ms, about 6,000 calls are made to function `getCaller()`, in a benchmarking invocation with reasonably large \( r \), both the number of prologue calls and the total prologue time are infinitesimally smaller than benchmarking calls and benchmark time.

We measured the average throughput in running \( r = 1,000 \) measurement sessions after \( m \) prologues runs. The results are depicted in Figure 5.8.1.
The picture portrayed by the figure is worrying; $m = 50$ prologue calls, whose total duration is a millisecond or two, were sufficient to make a very noticeable impact on an invocation whose duration is about 100 seconds. The figure demonstrates an increase by about 30%, from 15.2ns to 19.5ns. Further, a noticeable increase, albeit smaller is made even after $m = 3$ prologs calls.

Does this prologue effect ever wear out? Figure 5.8.2 plots our measurements after $m = 50$ prologue calls for increasing values of $r$.

We see in the figure that the effect of prologue runs does not wear out even after making $r = 128,000$ measurement sessions. (The duration of this last execution was about 4 hours.)
5.9 Discussion and Further Research

Section 5.5 showed, again, what many have previously observed—that a restart of the JVM, may yield different results. In attempt to study and understand this better (Section 5.6), we found that the problem, which is aggravated in more modern CPUs, is not entirely the blame of the JIT compiler as some may have thought before\(^2\); surprisingly, diversity and inconsistency of results are greater when the JIT compiler is disabled.

Delving deeper, we demonstrated that even a single JVM invocation may have multiple convergence points, and this multiplicity may even occur simultaneously. It may be argued that this multiplicity can be compensated for by sufficiently large restarts of the Java virtual machine. The concern is that micro-benchmarking of a certain code is used primarily to predict the impact of this code when integrated in a larger program. Brute force replication of the benchmark, without understanding the underlying issues, may yield results which are meaningless for an enveloping application program.

Worse, Section 5.8 demonstrated that the benchmarking results can be affected by prologue execution however infinitesimally short it is. This later finding, if not understood better (it could be a subtle artifact of “compilation planning”) may cast a doubt on the value of benchmarking for the evaluation of real programs.

Our works stops short of trying to understand the effects we saw: such a task requires an analysis of the deep stack of hardware and software abstractions. But, perhaps before such a study starts, more experimentation is required: one needs to examine the effects of other prologue executions, no-JIT runs on other machines, with, and without prologue executions, study other benchmarked code, with and without JIT, etc. Of course, judicious planning is an absolute necessity in finding a meaningful and effective research path in this exponentially huge design space.

\(^2\)see, e.g., J. Bloch’s “Performance Anxiety”, Devoxx 2010, http://www.parleys.com/#st=5&i=2103
Chapter 6

Summary

This thesis studied the integration of a relational algebra based query language into two mainstream object oriented programming languages – C++ and JAVA.

With regards to C++, we presented ARARAT, a language extension which seamlessly integrates relational algebra calculus into the language. One key aspect of ARARAT is type safety – both in the construction and composition of queries and in the definition of record types for storing query results. Another key aspect is smooth syntactic integration of relational algebra operators, which is achieved by using operator overloading. ARARAT represents queries as C++ expressions, allowing queries to be highly compositional and easily optimized.

As for JAVA, we presented several studies that were conducted in the course of extending the language with relational calculus.

First, an empirical study on the use of overloading in JAVA was described. in this study, we defined a taxonomy of uses of overloading, and classified overloaded methods and constructors in a large corpus of open source JAVA applications. The results of this study show that overloading, despite being extensively used, is usually not abused.

In another work, we offered a substitute to the most common use of overloading, which is emulation of default arguments. We extended the JAVA compiler by adding keyword parameters and default parameters to the language. This mechanism may also be used for integrating structural types into JAVA.

Finally, we presented a case study which explores inconsistencies encountered when we benchmarked a small JAVA code, and tries to characterize them.

As it turned out while working on this thesis, to fully define a programming language is a hard and complicated task. A language which involves several type
systems, with smooth interface between types, belonging to different systems, is even harder to define, and probably beyond the scope of a Ph.D. thesis.

In this work we presented our vision of an integrated OO language with database facilities, and the steps we took towards achieving this vision. Our hope is that a future work, based on these steps, as well as on related works that were published in the past few years, will integrate relational algebra into JAVA, while supplying a uniform model for accessing both internal and external data, as we initially planned to do. We believe that the nested relational algebra model is more natural to be included in JAVA than the flat model, since JAVA object may contain collections. Further, the nested model is also more natural to emulate SQL queries with GroupBy clauses.

Such an integrated language should handle the following aspects:

- Syntactic integration of relational algebra operators into JAVA, to allow direct representation of relational algebra expressions as expressions of the programming language.

- Introduction of structural types, for the representation of relational algebra records, and a smooth interface of these types with the underlying nominal type system. Whiteowk [47] addresses some of these issues.

- Type rules and type inference algorithms for relational algebra expressions.

- Introduction of declarative, loop free constructs for processing collections.

- Integration with features of the host language, such as subtyping, inheritance, encapsulation, classes and generics.
Bibliography


[26] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL—the Java tools language. In Tarr and Cook [121].


[99] Lajos Nagy and Ryan Stansifer. Polymorphic type inference for the relational algebra in the functional database programming language neon. In


[141] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and Eric Johnson. Adding domain-specific and general purpose language features to Java with the Java language extender. In Tarr and Cook [121], pages 728–729.


In Chapter 4, we present an extension of Java that allows for method parameters to be passed by name, and also supports default values. This extension is motivated by empirical research on the use of Java in Chapter 3, which showed that the addition of type constructors in Java 5 led to performance degradation, particularly when accessing fields and methods of more complex types.

In our experiments, we observed that even after controlling for factors known to affect performance, such as garbage collection, dynamic loading, and background processes of the system, we encountered inconsistent results.

Chapter 5 presents a case study where we aim to characterize the inconsistency we encountered.
Bamutul y'zordot ba'ret, ha-tfurseo mesfer beziv'ot mikrubot batehut, asher heketi ne'ur ha'verach.

In the absence of language support, the transformations to Groovy, a dialect of Java, were published. The work expanded the Java library by introducing functional constructs that hide iterative operations from the developer. In addition, Groovy supported other features designed for our work, such as operator overloading and collection evaluation. Thus, the language contained many practical and useful aspects that were supposed to be included in our work.

However, due to recent research, it became clear that most of the work we did with Java would be technical, and have added value. Therefore, the goal of expanding Java, to combine a type system and functional algebra, was not fully achieved. Nevertheless, during the work on this expansion, we explored several interesting directions.

The use of operator overloading in Java. The operators of functional algebra are polymorphic by nature. For example, it is possible to join two tables with the same attributes. Furthermore, the join operation can be performed on any two tables, regardless of their properties.

In Ararat, we extensively used operator overloading to implement this polymorphism. The overloading is a critical component in realizing our vision, but we did not want to add a feature that could be misused.

Therefore, we studied the use of this mechanism in Java and its potential misuse. This study was presented in Chapter 3.

To conduct this research, we developed a classification mechanism for the use of operator overloading and classification of occurrences of overloaded meth-
Stephen R. Drury was the first person to define a problem for which the answer was 'yes'.

The Java Virtual Machine (JVM) is used as the execution environment for Java programs. The virtual machine has two main components: the JVM itself and the Java class libraries. The JVM has a number of responsibilities, including loading classes, type checking, and interpreting bytecode. The Java class libraries contain classes for various tasks, such as networking, input/output, and graphics.

Java was designed to be a platform-independent language. It allows developers to write code that can run on any platform that supports the Java Virtual Machine. This is made possible by the use of the Java Class Libraries, which provide a standard set of classes for performing common tasks.

The Java Class Libraries are divided into several packages, each containing classes for a specific purpose. For example, the java.io package contains classes for input/output, the java.util package contains classes for collections, and the java.lang package contains classes for basic data types.

Java programs are typically compiled into bytecode, which is platform-independent and can be run on any platform that supports the Java Virtual Machine. However, some Java programs may be compiled into machine code for specific platforms, such as the Java ME (Mobile Edition) environment, which is designed for mobile devices.

Java supports a variety of programming paradigms, including object-oriented programming, procedural programming, and functional programming. It also supports multithreading and networking, allowing developers to create complex, concurrent applications.

Java has a rich set of features that make it a popular language for building enterprise applications. These features include a strong type system, garbage collection, and a rich set of class libraries. Java is also widely used in the development of mobile applications, games, and web applications.
מערכת תיומס ומכו. שפת היישום אחראית על ניהול הנתונים על ידי תכנית שנכתבה בשפה כדי. מתכנת רלציונים וasicsו absorbing בדרכו לכל.cm. שפת היישום SQL, Xpath, XQuery. מתכנת הצהרה מפורשת, מתכנת הצהרה בפורמט ציימי -方案 ציימי, ומקורות山村. המתכנת של הסיווגי נっこות נ쑹יין, ושFlowLayout הבוחר. התנונון של הסיווגי נסוורגי הוחלט, וسفيرת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה.

מערכת תיומס ומכו. שפת היישום אחראית על ניהול הנתונים על ידי תכנית שנכתבה בשפה כדי. מתכנת רלציונים וasicsו absorbing בדרכו לכל.cm. שפת היישום SQL, Xpath, XQuery. מתכנת הצהרה מפורשת, מתכנת הצהרה בפורמט ציימי- שפת מסד הנתונים מוט Cách לאו לו כל יום, והם מאותו מגרות וכלים מהמקורות של התוכנית, ושל התוכנית בשפת מסדי נתונים המבוססים על מה שמתכנת, של הסיווגי נסוורגי הוחלט, וسفيرת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה.

מערכת תיומס ומכו. שפת היישום אחראית על ניהול הנתונים על ידי תכנית שנכתבה בשפה כדי. מתכנת רלציונים וasicsו absorbing בדרכו לכל.cm. שפת היישום SQL, Xpath, XQuery. מתכנת הצהרה מפורשת, מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המבווה. מתכנת הצהרה בפורמט ציימי, או ולצורה ציימי, ובנגזרת הקול של השיחה המב覚え.
המเผย נעשיה בהנחיית פרופ' יוסף גל בפקולטה להנדסה להנדסת המחשב.

אני מודה לטכניון, לקרן הרבור ולתוכנית מלגות הדוקטורט של י.ב.מ, על התמיכה הכספית הנדיבה בהשתלמותי.

אני מאשר את כלiento, لكلור הרובר ולהצ potrà מתות הדוקטורט של י.ב.מ, על התVertexBuffer הנידובה בשטחימתי.
מנגנונים פנימיים וחיצוניים להרחבת שפות התוכנה

חיבור על מחקר

לשמ מיילר חלקי של הדרישות לקבלת התואר דוקטור לפילוסופיה

קרן לנץ

כרם לנץ

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
nces_8706 2012
נובמבר 2012
מנגנונים פנימיים וחיצוניים להרחבת שפה התכונות

קרוב לסיום