

JTL – the Java Tools Language

Tal Cohen Joseph (Yossi) Gil * Itay Maman†
The Technion—Israel Institute of Technology

Abstract

This paper gives an overview of JTL (the Java Tools Language, pronounced “Gee-tel”), a novel language for querying JAVA programs. JTL was designed to serve the development of source code software tools for JAVA, and for programming language extensions. Applications include definition of pointcuts for aspect-oriented programming, fixing type constraints for generic programming, specification of encapsulation policies, definition of micro-patterns, etc. We argue that the JTL expression of each of these is systematic, concise, intuitive and general.

JTL relies on a simply-typed relational database for program representation, rather than an abstract syntax tree. The underlying semantics of the language is that of queries formulated in First Order Predicate Logic augmented with transitive closure (FOPL*). Special effort was taken to ensure terse, yet very readable expression of logical conditions. The JTL pattern **public abstract class**, for example, matches all abstract classes which are publicly accessible, while **class { public clone(); }** matches all classes in which method `clone` is public. To this end, JTL relies on a DATALOG-like syntax and semantics, enriched with quantifiers and pattern matching which all but entirely eliminate the need for recursive calls.

The JTL processor includes a type inference engine. Also, JTL’s query analyzer gives special attention to the fragility of the “closed world assumption” in examining JAVA software, and determines whether a query relies on such an assumption. The performance of the JTL interpreter is comparable to that of JQuery after it generated its database cache, and at least an order of magnitude faster when the cache has to be rebuilt.

1. Introduction

The designers of frontier programming paradigms and constructs often choose, especially when static typing is an issue, to test bed, experiment, and even fully implement the new idea by an extension to the JAVA programming language. Examples include ASPECTJ [47], JAM [4], Chai [69], OpenJava [72], the host of type systems supported by pluggable type systems [5], and many more.

A prime component in the interaction of such an extension with the language core is the mechanism for selecting program elements to which the extension applies: The pointcuts of AspectJ, for example, select the point in the code on which an aspect is to be applied. Also, a key issue of implementing the conclusions of genericity

*Research supported in part by the IBM faculty award

†{ctal,yogi,imaman} @ cs.technion.ac.il

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

treatise [28,46] is checking whether a given set of classes are legible as parameters for a given generic construct—what Jarvi, Willcock, Lumsdaine et al. call a *concept*.

JTL (the *Java Tools Language*) is a declarative language, belonging in the logic-programming paradigm, designed for the task of selecting JAVA program elements. Two primary applications were in mind at the time of the initial design:

- (a) Join-point selection for aspect-oriented programming, where JTL can serve as a powerful substitute of ASPECTJ’s pointcut syntax, and,
- (b) expressing the conditions making up concepts, and in particular multi-type concepts, for use in generic programming.

In the course of developing the language it became clear JTL can be used not only for language extension, but also for other software engineering tasks, primarily as a tool to assist programmers understand the code they must modify. (The particular problem of program understanding, even if it is far from being entirely solved by JTL, is worthy of special mention since program development activities in industry include (and probably more and more so) the integration of new functionalities in existing code, and since maintenance remains a major development cost factor.)

JTL’s focus is on the modules in which the code is organized, packages, classes, methods, variables, including their names, types, parameters, accessibility level and other attributes. JTL can also inspect the interrelations of these modules, including questions such as which classes exist in a given unit, which methods does a given method invoke, etc. Additionally, JTL can inspect the imperative parts of the code by means of dataflow analysis. We currently work to extend JTL to deal with control flow as well.

1.1 Three Introductory Examples

JTL syntax is terse and intuitive; just as in AWK, one-line programs are abundant, and are readily wrapped within a single string. In many cases, the JTL *pattern* for matching a JAVA program element looks exactly like the program element itself. For example, the JTL *predicate*¹

```
public abstract void ()
```

matches all methods (of a given class) which are abstract, publicly accessible, return **void** and take no parameters. Thus, in a sense, JTL mimics the *Query By Example* [75] idea.

As in the logic paradigm, a JTL program is a set of predicate definitions, one of which is marked as the program goal.

Even patterns which transcend the plain JAVA syntax should be understandable, e.g.,

¹The terms “predicate” and “pattern” are used almost interchangeably; “pattern” usually refers to a unary predicate.

```

abstract class {
    [long | int] field;
    no abstract method;
}

```

matches abstract classes in which there is a field whose type is either **long** or **int** and no abstract methods.

The first line in the curly brackets is an existential quantifier ranging over all class members. The second line is a negation of an existential quantifier, i.e., a universal quantifier in disguise, applied to this range.

JTL can also delve into method bodies, by means of intra-procedural dataflow analysis, similar to that of the class file verifier. Consider for example *cascading methods*, i.e., methods which can be used in a cascade of message sends to the same receiver, as in the following JAVA statement

```
(new MyClass()).f().g().h();
```

in which *f*, *g* and *h* are methods of *MyClass*. Then, the following JTL pattern matches all cascading methods:

```

instance method {
    all [ !returned | this ];
}

```

The curly brackets in the above pattern denote the set of values (including temporaries, parameters, constants, etc.) that the method may generate or use. The statement inside the brackets is a requirement that all such values are either not returned by the method, or are provably equal to **this**, and therefore guarantees that the only possible value that the method may return is **this**.

1.2 Applications

The JTL interpreter (work on the compiler is in progress) can be used in two ways: (i) as a stand-alone program; (ii) as a JAVA library, an API, to be called directly from within any JAVA program, in a manner similar to SQL. This API makes JTL useful in implementing not only language extensions (such as a pointcut evaluation engine for an AOP compiler), but also in various kinds of software engineering tools, including searches in programs [56], LINT-like tools [26, 44, 64], and pattern identification [30].

JTL's ability to generate output text based on matched program elements enables the use of JTL not only for *search* operations in programs, but also for *search-and-replace* operations. A key applications for this ability is the context-aware renaming of program elements [27].

The brevity of expression in JTL makes it suitable for integration as an interactive search language in an IDE. Also, a JTL configuration file can specify to a compiler program elements on which special optimization effort should be made, exclusion from warnings, etc. Thus, JTL's unique properties as a "tool for making tools", are its suitability for direct user interaction, and begin a small language for configuring other tools.

To an extent, JTL resembles TCL [60], the general purpose Tool Command Language, except that JTL concentrates on language processing.

In evaluating JTL suitability for other applications we wrote several collections of JTL patterns, the two largest being (i) the implementation of the entire set μ -patterns developed by two of us [30] and (ii) the implementation of the entire set of Eclipse² and PMD³ warning messages (with the exclusion of warnings pertaining to source only, since the current implementation of JTL works on compiled classes). Other applications in which we used JTL were aspect pointcuts, and pre-conditions for mixins and generics.

²<http://www.eclipse.org>

³<http://pmd.sourceforge.net>

An Eclipse plugin for running JTL queries over JAVA projects was implemented, and its performance was compared with that of a similar plugin which uses JQuery [45] for querying JAVA code.

1.3 Underlying Model

Underlying JTL is a *conceptual* representation of a program in a simply-typed relational database. The JTL user can think of the interrogated JAVA program as a bunch of program elements stored in such a database.

JTL is declarative, sporting the simple and terse syntax and semantics of logic programming for making database queries augmented with a number of enhancements that make it even more concise and readable. Predicates are the basic programming unit. The language features a set of native predicates (whose implementation is external to the language) augmented with a library of pre-defined predicates built on top of the native ones. Many of the native and pre-defined predicates are conveniently named after JAVA keywords.

Thus, the scheme of this database is defined by the set of native JTL predicates. Standard library predicates and user-defined predicates, defined on top of the natives, can be thought of as database *views* or *pre-defined queries*.

Interestingly, the nature of JAVA in particular, and of many other software systems in general, is that unlike traditional database systems, the conceptual database that JTL uses is *infinite*. In almost all programming languages, one cannot hope to obtain all user code which uses a certain code bit stored in a software library. Similarly, the list of classes that inherit from a given class is unbounded.

The JTL processor analyzes the queries presented to it, determining whether they are open-ended, i.e., the size of the result they return is unbounded. (In practice, only a finite approximation of the infinite database is stored. An open-ended query can be thought as a query whose size increases indefinitely with that of the approximation.)

Note that this conceptual representation does not dictate any concrete representation for the JTL implementation. JTL is applicable to several formats of program representation, ranging from program source code, going through AST representations, JAVA reflection objects, BCEL⁴ library entities, to strings representing the names of program elements. In fact, JTL's JAVA API is characterized by both input- and output- data representation flexibility, in that JTL calls can accept and return data in a number of supported formats.

We stress that JTL can be implemented in principle on top of any source code parser, including the JAVA compiler itself.

Two central concerns in the language design were the *simplification* of the idiosyncracies of logic programming, and *scalability*. JTL thus features specific constructs for set manipulation, quantification and other means that eliminate much of the need for loops (recursive calls in the logic programming world). As a result, unlike DATALOG [15] and PROLOG queries, JTL predicates are defined by a single rule, written in a handful of lines, and often is a single line.

Also (although not discussed in much detail here), JTL supports modularity, name spaces, documentation (similar to that of JavaDoc⁵) and indexing (similar to that of EIFFEL).

Underlying JTL's semantics is first order predicate logic with no function symbols and augmented with transitive closures, denoted FOPL*. The first order logic represented by JTL is restricted to finite structures (assuming a given database approximation). It is computationally equivalent to DATALOG with stratified negation [73], and enjoys the same theoretical advantages of the formalism, including *modular specification*, *polynomial time complexity*,

⁴<http://jakarta.apache.org/bcel>

⁵<http://java.sun.com/j2se/javadoc>

and a wealth of *query optimization techniques* [36].

Indeed, the JTL compiler under development will generate DATALOG output for an industrial-strength DATALOG engine.

Even though termination is always guaranteed (on a finite database), it is a basic property of FOPL* that other questions are undecidable. For example, it follows from Gödel's incompleteness theorem that it is impossible *in general* to determine e.g., if two queries are equivalent, a query is always empty, the results of one query is contained in another, etc. These limitations are not a major hurdle for most JTL applications. Moreover, there are textbook results [13] stating that such questions are decidable, with concrete algorithms, if the use of quantifiers is restricted, as could be done for certain applications.

1.4 Contribution

We believe that JTL will be useful for many tool writers, and as an IDE add-on, especially considering its search-and-replace capabilities. Beyond this, the contributions of this work is in demonstrating that a simple query-by-example like syntax is possible for many tasks of querying OO programs, and in showing that this syntax stands on a solid theoretical ground. It may be possible to put together a JAVA-like syntax for JAVA queries in an ad hoc fashion. The challenge we took upon ourselves was the combination of the sound underlying computational model and the query-by-example front end. Also, in using a DATALOG-based model (and in contrast with PROLOG as some other recent tools do) we achieve a termination guarantee, and the wealth of theory on database query optimization for concrete scalable implementation.

As the name suggests, JTL is specific to JAVA. The bountiful class file format of JAVA, its extensive documentation and the verification process, made it possible to carry out the JTL processing on the binary- rather than on the source representation of a program. The issues involved in generalizing JTL ideas to other programming languages and software model representations should become clearer with the more detailed exposition.

The JTL semantic model may also be of interest. Perhaps surprisingly, queries of programs written in languages with runtime loading (such as JAVA) and of program libraries, are naturally described as running over an *infinite* database. We developed an algorithm (whose full description is beyond the scope of this paper) for determining whether such queries return a finite or an infinite result. This algorithm also helps in finding a *top-down* efficient evaluation scheme for finite JTL queries, and can decide whether a given static analysis algorithm (if formulated as a logic programming query) makes a whole-world analysis assumption.

Outline. There is a delicate balance to strike in presenting a new language such as JTL. On the one hand, it is tempting to quickly get to the interesting applications; on the other, it is difficult to appreciate the code samples, which may even be mistaken as a *deus ex machina*, i.e., a bunch of hacks, without seeing first the underlying semantics. Worse, programming languages are usually introduced in two independent texts: a language tutorial supplying the intuition and a user's guide with the precise definitions. Since some of the novel ideas of JTL require both an intuitive and a precise presentation, we needed to juggle a bit of both here.

We chose to start with a brief language tutorial, in Sec. 2, which shows how JTL can be used to inspect the non-imperative aspects of JAVA code, i.e., everything but the method bodies. Queries of the imperative aspects of the code are the subject of Sec. 3.

A more detailed explanation of the semantics, including a discussion of the infinite database model is then presented in Sec. 4. This section also expands the discussion a bit, towards a theory of searches in databases of object oriented software.

Readers who are more interested in actual code may choose to leap directly to Sec. 5, which presents some more advanced applications and code samples.

Sec. 6 compares the performance of JTL with that of JQuery, a JAVA query tool which uses a similar underlying paradigm. Sec. 7 then discusses some of the issues involved in porting JTL to other programming languages, specifically C#. Sec. ?? elaborates further on the motivation, surveying existing systems and discusses potential applications. Finally, Sec. 9 concludes.

2. The JTL Language

This section gives a brief tutorial of JTL, assuming some basic familiarity with logic programming. The text refers only to the inspection of the non-imperative aspects of the software. The next section will build upon this description the mechanism for delving into method bodies. Readers interested more in the underlying semantics are encouraged to skip to Sec. 4, returning here only for reference.

The main issues to note here are the language *syntax*, in which a JAVA program element is matched by a JTL pattern which is very similar in structure to that element (see Sec. 2.1), and the *extensions* to the logic paradigm, specifically, arguments list patterns (Sec. 2.2), transitive closure standard predicates (Sec. 2.3), and quantifiers (Sec. 2.4) which make it possible to achieve many programming tasks without recursion.

The two most important data types, what we call *kinds*, of JTL are (i) **MEMBER**, which represents all sorts of class and interface members, including function members, data members, constructors, initializers and static initializers; and (ii) **TYPE**, which stands for JAVA **classes**, **interfaces**, and **enums**, as well as JAVA's primitive types such as **int**.

Another important kind is **SCRATCH**, which, as the name suggests, stands for a temporary value used or generated in the course of computation of a method. Scratches originate from a dataflow analysis of a method, and are discussed below at Sec. 3.

A JTL program is a set of definitions of named logical *predicates*. Execution begins by selecting a predicate to execute as a *query*.

As in PROLOG, predicate names start with a lower-case letter, while *variables* and parameters names are capitalized. Identifiers may contain letters, digits, or an underscore. Additionally, the final characters of an identifier name may be "+" (plus), "*" (asterisk), or "'" (single quote).

2.1 Simple Patterns

Many JAVA keywords are native patterns in JTL, carrying essentially the same semantics. For example, the keyword **int** is also a JTL pattern **int**, which matches fields of type **int** and methods whose return type is **int**. The pattern **public** matches all **public** program elements, including public class members (e.g., fields) and public classes. Henceforth, our examples shall use these keywords freely; no confusion should arise.

Not all JTL natives are JAVA keywords. A simple example is **anonymous**, defined on **TYPE**, which matches anonymous classes.

Some patterns (like **abstract**) are overloaded, since they are applicable both to types and members. Others are monomorphic, e.g., **class** is applicable only to **TYPE**.

Another example is pattern type, defined only on TYPE, which matches all values of TYPE. This, and the similar pattern member (defined on MEMBER) can be used to break overloading ambiguity.

JTL has two kinds of predicates: *native* and *compound*. Native predicates are predicates whose implementation is external to the language. In other words, in order to evaluate native predicates, the

JTL processor must use an external software library accessing the code. Native patterns hence are declared (in a pre-loaded configuration file) but not defined by JTL.

In contrast, compound patterns are defined by a JTL expression using logical operators. The pattern

```
public, int (2.1)
```

matches all **public** fields of type **int** and all **public** methods whose return type is **int**. As in PROLOG, conjunction is denoted by a comma. In JTL however, the comma is optional; patterns separated by whitespace are conjuncted. Thus, (2.1) can also be written as **public int**.

As a matter of style, the JTL code presented henceforth uses mostly whitespace for conjunction; commas are used for readability—breaking long conjugation sequences into subsequences of related predicates; Disjunction is denoted by a vertical bar, while an exclamation mark stands for logical negation. Thus, the pattern

```
public | protected | private
```

matches JAVA program elements whose visibility is not default, whereas **!public** matches non-**public** elements.

Logical operators obey the usual precedence rules, i.e., negation has the highest priority and disjunction has the lowest. Grouping is by square parenthesis, as in

```
!private [byte|short|int|long]
```

which matches non-**private**, integral-typed fields and methods.

A *pattern definition* names a pattern. After making the following two definitions,

```
integral := byte | short | int | long;
enumerable := boolean | char;
```

the newly defined patterns, *integral* and *enumerable*, can be used anywhere a native pattern can be, as in e.g.,

```
discrete := integral | enumerable
```

Beyond the natives, JTL has a rich set of pre-defined *standard* patterns, including patterns such as *integral*, *enumerable*, *discrete* (as defined above), *method*, *constructor* (both with the obvious semantics) *extendable* := **!final** type, *overridable* := **!final !static** method, and many more.

2.2 Signature Patterns

Signature patterns pertain to (a) the name of classes or members, (b) the type of members, (c) arguments list, (d) declared thrown exceptions, and (e) annotations (meta-data).

Name Patterns. A *name pattern* is a regular expression preceded by a single quote, or a previously-declared name. Standard JAVA regular expressions⁶ are used, except that the wildcard character is denoted by a question mark rather than a dot. Name literals and regular expressions are quoted with single quotes. The closing quote can be omitted if there is no ambiguity.

For example, **void** 'set[A-Z]?*' method matches any **void** method whose name starts with “set” followed by an upper-case letter.

If the name pattern does not contain any regular expression operators, as in

```
toString_p := 'toString method; (2.2)
```

then the pattern can be made clearer by using a **name** statement to declare *toString* as a member name and get rid of the quote. Thus, an alternative definition of (2.2) is

```
name toString;
toString_p := toString method; (2.3)
```

⁶as defined by `java.util.regex.Pattern`.

*In truth, the above is redundant, since an implicit **name** statement pre-declares all methods of the JAVA root class `java.lang.Object`.*

Type Patterns. *Type patterns* make it possible to specify the JAVA type of a non-primitive class member. A type pattern is a regular expression preceded by a forward slash, e.g., pattern `/java.util.?*/` method matches all methods with a return type from the `java.util` package or its sub-packages. The closing slash is optional.

The distinction between type patterns and name patterns only makes sense for members. In matching types, there is no such distinction, and both kinds of literals can be used. Appending an asterisk to a type pattern makes it match subclasses as well (including sub-interfaces and implementing classes). To match proper subclasses only, use a plus sign.

The forward slash is not necessary for type names which were previously declared as such by a **typename** declaration. For example,

```
typename java.io.PrintStream;
printstream_field := PrintStream field; (2.4)
```

matches any field whose type is `java.io.PrintStream`.

All the types (including classes, interfaces and enumerations) declared in the `java.lang` package are pre-declared as type names, including `Object`, `String`, `Comparable`, and the wrapper classes (`Integer`, `Byte`, `Void`, etc.).

Here is a redefinition of *toString_p* pattern (2.3), which ensures that the matched method returns a `String`.

```
toString_p := String toString method; (2.5)
```

Arguments List Patterns. JTL provides special constructs which all but eliminate recursion. An important example is *arguments list patterns*, used for matching against elements of the list of arguments to a method. (Internally, such lists are stored in a linked list, using standard PROLOG-like head and tail relations.)

The most simple argument list is the empty list, which matches methods and constructors that accept no arguments. Here is a rewrite of (2.5) using such a list:

```
toString_p := String toString();
```

(Note that the above does not match fields, which have no argument list, nor constructors, which have no return type.)

An asterisk (“*”) in an arguments list pattern matches a sequence of zero or more types. Thus, the standard pattern

```
invocable := (*);
```

matches members which may take any number of arguments, i.e., constructors and methods, but not fields, initializers, or static initializers. An underscore (“_”) is a single-type wildcard, and can be used in either the argument list or in the return type. Hence,

```
public _ (_, String, *); (2.6)
```

matches any public method that accepts a `String` as its second argument, and returns any type. (Constructors cannot be matched by (2.6), since they have no return type.)

Other Signature Patterns. There are patterns for matching the **throws** clause of the signature, e.g.,

```
io_method := method
            throws /java.io.IOException;
```

There are also patterns which test for the existence or absence of specific annotations in a class, a field or a method, and for annotation values. For example, the following pattern will match all methods that have the `@Override` annotation:

```
@Override method
```

These are not discussed here in detail further in respect of space limitations.

2.3 Variables

It is often useful to examine the program element which is matched by a pattern. JTL employs variable binding, similar to that of PROLOG, for this purpose. For example, by using variable *X* twice, the following pattern makes the requirement that the two arguments of a method are of the same type:

```
firstEq2nd := method (X,X);
```

Similarly, the pattern

```
return_arg := RetType (*, RetType, *);
```

matches any method whose return type is the same as the type of one of its arguments.

Predicates. Patterns are parameterless predicates. In general, a predicate may take parameters. As usual in logic programming, *parameters* are nothing more than externally accessible variables. Consider for example the predicate

```
is_static[C] := static field _:C; (2.7)
```

which takes parameter *C*. When invoked with a specific value for *C*, pattern *is_static* matches only **static** fields of that exact type.

Conversely, if the predicate is invoked without setting a specific value for *C*, then it will assign to *C* the types of all **static** fields of the class against which it is matched. The semantics by which a parameter to a predicate can be used as *either* input or output is standard in logic programming; the different assignments to *C* are made by the evaluation engine, in the process of trying to satisfy the predicate.

Note however that since JTL uses a database-, DATALOG-like semantics, rather than the recursive evaluation engine of PROLOG, each type *C* satisfying (2.7) will show only once in the output, even if there two or more fields of that type.

(Because square brackets denote parameter passing, array types in JTL must be preceded by a slash; for example, `/int []` field will match any field of type `int []`.)

Native Predicates. JTL has several native parameterized predicates. The names of many of these are JAVA keywords. For example, predicate `implements [I]` holds for all classes which implement directly the parameter *I* (an **interface**).

This is the time to note that the predicates `implements [I]` and `is_static[C]`, just as all other patterns presented so far, have a hidden argument, the *receiver*, which can be referenced as **This** or #.

Other native predicates of JTL include `members[M]` (true when *M* is one of **This**'s members, either inherited or defined), `defines[M]` (true when *M* is defined by **This**), `overriding[M]` (true when **This** is a method which overrides *M*), `inner[C]` (true when *C* is an inner class of **This**), and many more.

The following example shows how a reference to **This** is used to define a pattern matching what is known in C++ jargon as "copy constructors":

```
copy_ctor := constructor(T), T.members[This];
```

This example also shows how a predicate can be invoked with a specific receiver, by using a JAVA-like dot notation. The default receiver is **This**, similar to `this` being the default receiver in JAVA method invocations.

The copy_ctor predicate works like this: first, the pattern constructor(T) requires that the matched item, i.e., This, is a constructor, which accepts a single parameter of some type T. Next, T.members[This] requires that This—the matched constructor—is a member of its argument type T, or in other words, that the constructor's accepted type is the very type that defines it.

Literals, just as variables, can be used as actual parameters. For

example, `class implements[M]` matches any class that implements interface *M*, whereas

```
interface extends [/java.io.Serializable]
```

matches any interface that extends the `Serializable` interface.

The square brackets in an invocation of a named predicate are optional, since JTL knows predicate arity. The above could thus have also been written as:

```
interface extends /java.io.Serializable
```

Standard Predicates. JTL also has a library of standard predicates, many of which are defined as transitive closure of the native predicates. Fig. 2.1 shows a sample of these.

Fig. 2.1 Some of the standard predicates of JTL

```
inherits[M] := members[M] !defines[M];
container[C] := C.members[This];
precursor[M] := M.overriding[This];
implementing[M] := !abstract,
overriding[M] M.abstract;
abstracting[M] := abstract,
overriding[M] !M.abstract;
extends+[C] := extends[C] |
extends[C'] C'.extends+[C];
extends*[C] := C is This | extends+[C];
interfaceof[C] := C.class C.implements[This];
interfaceof+[C] := C.implements+[This];
interfaceof*[C] := C.implements*[This];
```

The figure makes apparent the JTL naming convention by which the reflexive transitive closure of a predicate *p* is named *p**, while the anti-reflexive variant is named *p+*. The myriad of recursive definitions such as these saves much of the user's work; in particular it is rare that the programmer is required to employ recursion.

It is interesting to examine the "recursive" definition of one of these predicates, e.g., `extends+`:

```
extends[C] | extends[C'] C'.extends+[C]
```

It may appear at first that with the absence of a halting condition, the recursion will never terminate. A moment's thought reveals that this is not the case. Since JTL uses a bottom-up construction of facts, starting at a fixed database, the semantics of this recursive definition is not of stack-based recursive calls, but rather as dynamic programming, or work-list, approach for generating facts.

Predicate Name Aliases. The name `extends+` suggests that it is used as a verb connecting two nouns. As mentioned above, we can even write

```
C extends+ C'
```

But, the same predicate can be used in situations in which, given a class *C*, we want to generate the set of *all* classes that it extends. A more appropriate name for these situations is `ancestors`. It is possible to make another definition

```
ancestors[C] := extends+[C];
```

To promote meaningful predicate names, JTL offers what is known as *predicate name aliases*, by which the same predicate definition can introduce more than one name to the predicate. Aliases are written as an *definition annotation* which follows the main rule. The definition of `extends+` has such an alias

```
extends+[C] := extends C |
extends C', C'.extends+[C];
Alias ancestors;
```

The use for an alias named `ancestors` will become clear when predicate (2.9) is presented, below.

Native predicates can also have aliases, which are specified along with their declaration.

2.4 Queries

As mentioned previously, JTL’s expressive power is that of FOPL[†]. Although it is possible to express universal and existential quantification with the constructs of logic programming, we found that the alternative presented in this section is more natural for the particular application domain.

Consider for example the task of checking whether a JAVA class has an `int` field. A straightforward, declarative way of doing that is to examine the set of all of the class fields, and then check whether this set has a field whose type is `int`.

The following pattern does precisely this, by employing a *query* mechanism:

```
has_int_field := class members: {
    exists int field; (2.8)
};
```

Here, the query `members: { $Q_1; \dots; Q_n$ }` generates first the set of all possible members M , such that `#.members[M]` holds. (The “`members:`” portion of the query is called the *generator*.)

This set is then passed to each of $Q_1; \dots; Q_n$, the *quantifiers* embedded in the curly brackets. The query holds if all of these quantifiers hold for this set.

In (2.8), there was only one quantifier: the JTL statement `exists int field` is an existential quantifier which holds whenever the given set has an element which matches the pattern `int field`.

The next example shows two other kinds of quantifiers.

```
class ancestors: {
    all public; (2.9)
    no abstract;
};
```

The evaluation of this pattern starts by computing the generator. In this case, the generator generates the set of all classes that the receiver `extends` directly or indirectly, i.e., all types C for which `#.ancestors[C]` holds. (Recall that `ancestors` is an alias for `extends+`, defined above.) The first quantifier checks whether all members of this set are `public`. The second quantifier succeeds only if this set contains no `abstract` classes. Thus, (2.9) matches classes whose superclasses are all public and concrete.

JTL features several other kinds of quantifiers: `many p` holds if the queried set has two or more elements for which pattern p holds; whereas `one p` holds if this set has precisely one such element.

The existential quantifier is the most common; hence the `exists` is optional. Also, a missing generator defaults to `members:`. Hence, a concise rewrite of (2.8) is

```
has_int_field := class {
    int field; (2.10)
};
```

In the two examples shown here, the generator was a predicate with a single named parameter and an implicit receiver. In such cases, the generator generates a set of primitive values, which are the possible assignments to the argument. However, in general, the generator generates a relation of named tuples, and the quantifiers are applied to the set of these tuples. We discuss the underlying semantics of queries in greater detail in Sec. 4.

3. Queries of Imperative Code

The executional aspect of JAVA code remained beyond the description of JTL in the previous section. This aspect is primarily method bodies, but also other imperative code, including constructors, field initializers and static initializers.

Now that the bulk of the language syntax is described, we can turn to the question of queries of *imperative entities*. To an extent, queries of these entities is mostly a matter of library design rather

than a language design. Recall that JTL native predicates are implemented as part of the supporting library that the JTL processor uses for inspecting JAVA code. Extending this library, without changing the JTL syntax, can increase the search capabilities of the language.

Sec. 3.1 shows how by adding a set of native predicates, JTL *can* be extended to explore an abstract syntax tree representation of the code. This section also explains why this extension was not implemented yet. We chose instead to implement a pedestrian set of natives that make it possible to explore the fields and methods that executional code uses, as described in Sec. 3.2. The more sophisticated mechanism that JTL uses for inspecting method bodies is through a dataflow analysis, as described in Sec. 3.3.

3.1 Abstract Syntax Trees and JTL

Executional code can be represented by an abstract grammar, with non-terminal symbols for compound statement such as `if` and `while`, for operations such as type conversion, etc. One could even think of several different such grammars, each focusing on a different perspective of the code.

Code can be represented by an abstract syntax tree whose structure is governed by the abstract grammar. To let JTL support such a representation, we can add a new kind, `NODE`, and a host of native relations which represent the tree structure. For example, a native binary predicate `if` can be used to select `if` statement nodes and the condition associated with it; a binary predicate `then` can select the node of the statement to be executed if the `if` condition holds; another binary predicate, `else`, may select the node of the statement of the other branch, etc.

As an example of an application for such a representation, consider a search for code fragments such as

```
if (C)
    return true;
else return false;
```

with the purpose of recommending to the programmer to write “`return C;`” instead. The following pattern matches such code:

```
boolean_return_recommendation :=
    if[_] then[S1] else[S2],
    S1.return[V1], S2.return[V2],
    V1.literal["true"],
    V2.literal["false"];
```

The above pattern should be very readable: we see that its receiver must be a `NODE` which is an `if` statement, with a don’t-care condition (i.e., `_`), which branches control to statements $S1$ and $S2$; also both $S1$ and $S2$ must be `return` statements, returning nodes $V1$ and $V2$ respectively. Moreover, the patterns requires that nodes $V1$ and $V2$ are literal nodes, the first being the JAVA `true` literal, the second a `false`.

In principle, such a representation can even simultaneously support more than one abstract grammar. Two main reasons stood behind our decision not to implement, or even define (at this stage), the set of native patterns required for letting JTL explore such a representation of the code.

1. *Size.* Abstract grammars of JAVA (just as any other non-toy language) tend to be very large, with tens and hundreds of non-terminal symbols and rules. Each rule, and each non-terminal symbol, requires a native definition, typically more than one. The effort in defining each of these is by no means meager.
2. *Utility.* Clearly, an AST representation can be used for representing the non-imperative aspects of the code. The experience gained in using the non-AST based representation of JTL for exploring these aspects, including type signatures,

declaration modifiers, and the interrelations between classes, members and packages, indicated that the abstraction level offered by an abstract syntax tree is bit low at times.

A third, (and less crucial) reason is that it is a bit difficult (though not infeasible) to elicit the AST from the class file, the data format used in our current implementation.

3.2 Pedestrian Code Queries

In studying a given class, it is useful to know which methods use which fields. The following JTL pattern, for example, implements one of Eclipse’s warning situations, in which a private member which is never used.

```
unused_private_member := private field,
  This Is F,
  declared_in C, C inners*: {
    all !access[F];
  }
```

The pattern fetches the class C that defines the field, and then uses the reflexive and transitive closure of the inner relation, to examine C, its inner classes, their inner classes, etc., to make sure that none of these reads or writes to this field. (The unification (**This is F**) is for making the receiver field accessible inside the curly brackets.)

The pattern `access` is defined in the JTL library. The definition, along with some of the other standard patterns that can be used in JTL for what we call *pedestrian code queries* is shown in Fig. 3.1. Such queries view the method body as an unordered set of bytecode instructions, checking whether this set has certain instructions in it.

In the figure we see that the definition of `access` is based on the overloaded predicates `read` and `write`. The native predicate `read[F]` (respectively `write[F]`) holds if the receiver is a method whose code reads (respectively writes to) the field F. The second (respectively the third) line of the figure, overloads the native definition of `read` (respectively `write`), so that it applies also to receivers whose kind is **TYPE**.

Fig. 3.1 Standard predicates for pedestrian code queries

```
access[F] := read F | write F; Alias accesses;
read[F] := offers M, M read F; Alias reads;
write[F] := offers M, M read F; Alias writes;

calls[M] := invokes_interface[M] |
  invokes_virtual[M] |
  invokes_static[M] |
  invokes_special[M];
Alias invokes, invoke;

use[X] := access[M] | invoke[M]; Alias uses;
```

The figure also makes uses of the four other pedestrian natives for inspecting code: `invokes_interface`, `invokes_virtual`, `invokes_static`, and `invokes_special`. (These natives also have aliases identical to the bytecode mnemonics.)

With this minimal set of six natives, several interesting patterns can be defined. For example, predicate

```
creates[T] := invokes_static[M], M.ctor,
  M.declared_in[T];
```

is true when the receiver creates an object of type T. Also, the following predicates test whether a method *refines* its precursor

```
refines[M] := overrides[M] invokes_special[M];
refines := refines[_]; Alias refiner;
```

The following predicate checks whether a method is not empty

```
does_something := !void | invokes[_] |
```

```
writes[_];
```

(If a method does not return a value, does not invoke any other method, nor write to a field, then it must have no meaningful effect.) With the above, we implemented an interesting PMD rule, signalling an unnecessary constructor, i.e., the case that there is only one constructor, it is public, has an empty body, and takes no arguments.

```
unnecessary_constructor := class {
  constructor => public () !does_something;
}
```

The following predicate identifies a case that a constructor calls another constructor C of class T.

```
c_call[C,T] := constructor invokes_special[C]
  C.constructor C.declared_in[T]
```

With this predicate, we can present three rules which identify the three different ways that a constructor may begin its mission in JAVA.

```
c_delegation[C] := // First line is "this(...)"
  declared_in[MyClass] c_call[C,MyClass];
c_refinement[C] := // First line is "super(...)"
  declared_in[MyClass] c_call[C,Parent],
  MyClass.extends[Parent];
c_handover[C] := // Either
  ctor_delegation[C] | ctor_refinement[C];
```

3.3 Dataflow Code Queries

As a substitute to AST queries and at a higher level of abstraction than the pedestrian queries, stand dataflow code queries. In the course of execution of imperative entities many temporary values are generated. Dataflow analysis studies the ways that these values are generated and transferred. The idea is similar to dataflow analysis as carried out by an optimizing compiler [1, Sects. 10.5–10.6], or by the JAVA bytecode verifier [50, Sec. 4.92].

3.3.1 Scratches for Dataflow Analysis

To implement dataflow analysis, we introduce a new JTL kind, **SCRATCH**, which represents what is called in the compiler lingo a “variable definition”, i.e., an assignment to a temporary variable. In JAVA we find two categories of temporary variables on what is called a “method frame”:

1. A location in the “local variables array”, including the locations reserved in this array for method parameters, and in particular the “**this**” parameter of instance methods, as well as local variables that an imperative entity may define.⁷
2. A location in the “operands stack”, used for storing temporary variables in the course of the computation.

As usual with dataflow analysis, there is a fresh scratch for each assignment to a temporary variable. Also, scratches are generated on a merge of control flow. A scratch is anonymous; it does not carry with it its location in the frame.

An assignment to a scratch can be either from one of the following sources: another scratch, an input parameter value, a constant, a field, a value returned from a method or a code entity, an arithmetical operation, or a thrown exception. A scratch can be assigned to another scratch, passed as a parameter, assigned to a field, thrown or returned.

The dataflow analysis is implemented at the class file level. Obviously, this can only be done with a non-optimizing compiler, which

⁷Note that the verification process guarantees that we can treat two adjacent locations which are used for storing a **long** or **double** variables.

makes a lossless translation of the source dataflow into an equivalent dataflow of the intermediate or target language. Luckily, the standard JAVA compiler obeys this requirement.

It should be clear that the dataflow information and analysis we carry is also feasible by starting from the source level, generating temporaries for all intermediate values.

On the other hand, it should be noted that JAVA semantics and tradition also helped in making our dataflow analysis more effective; among the contributing factors we can mention the tendency to use short methods, the requirement that all locals are initialized before they are used, call-by-value semantics, no pointer arithmetic, our decision to ignore arrays, etc.

3.3.2 Using Scratches

The following pattern, capturing the authors' understanding of the notion of *setter*, gives a quick taste of the manner of using data flow information in JTL.

```
setter := void instance method(_) !calls[_] {
  putfield[_] => parameter;
  one putfield[_];
  putfield[_] Ref.this.
  no [ putstatic[_] | get[_] | compared ];
};
```

The first line in the above requires that the receiver is a **void** instance method, taking a single parameter, and (by using a pedestrian predicate) that it calls no other methods. The predicates in the curly brackets make the following requirements in order: (i) all assignments to a field are of a scratch that is provably equal to the single parameter of the method; (ii) there is precisely one assignment to an instance field in the method; (iii) this assignment is to a field using an object reference which is provably equal to the **this** implicit parameter of the method; and that (iv) there are no assignments to a static field, nor field retrievals, nor a comparison in the method.

The example shows that the formulation of data flow requirements is not so simple. Moreover, the precise notion of a setter is debateable. We argue that JTL's terse, English-like syntax, augmented with the natural coding of work-list algorithms in the logic paradigm, help in quick testing and experimenting with patterns such as the above.

Dataflow analysis is a large topic, and its application in JTL involves about three dozens of predicates. We can only give here a touch of the structure of JTL's predicates library and the patterns that can be written with it.

Tab. 1 lists the essential native unary predicates defined on scratches. The predicates are obtained by a standard (conservative) dataflow analysis similar to that of the verifier does. Thus, predicate *parameter* holds for all scratches which are provably equal to a parameter, *nonnull* for temporaries which are provably non-null, etc.

Table 1: Native unary predicates of scratches

Predicate	Meaning
<i>parameter</i>	a method parameter stored in the LVA
<i>constant</i>	a constant
null	the null constant
<i>nonnull</i>	cannot be the null constant
<i>temp</i>	an operand-stack scratch
this	equal to parameter 0 of an instance method
<i>local</i>	an (uninitialized) automatic variable in the LVA
<i>returned</i>	returned by the code
<i>athrow</i>	thrown by the code
<i>caught</i>	obtained by catching an exception
<i>compared</i>	compared in the code

The native binary predicate *scratches*[S] (also aliased as *has_scratch*) holds if S is a scratch of the method **This**, and serves as the default generator of methods. Hence, the curly bracket in the pattern

```
instance method {
  returned => this;
}
```

iterate over all scratches of a given method, checking that every scratch returned by the method is equal to the **this** parameter. Also, the binary predicate *typed*[T] holds if T is the type of the scratch # (**This**). The following pattern returns the set of all types that a method uses:

```
use_types[T] := method has_scratch[S]
  S.typed[T];
```

The most important predicate connecting scratches is *from*[S], which holds if scratch S is assigned to scratch #. Similarly, *func*[S] holds if # is computed by an arithmetical computation. We also have the definition

```
depend[S] := func[S] | from[S];
```

As usual, *from**, *func** and *depend** denote the reflexive transitive closure of *from*, *func* and *depend*.

There is also a native predicate for each of the four bytecode instructions used for accessing fields. For example, the binary predicate *putstatic*[F] holds if the scratch # is assigned to **static** field F, while *getfield*[F,S] holds if # is retrieved from field F with scratch S serving as object reference.

In addition, there are two predicates for each of the instructions for method calling, e.g., a predicate *invokespecial*[M,S] holds if scratch # is used as an argument to a call of method M, where scratch S serves as the receiver, and a predicate *get_invokestatic*[M] which holds if # obtains its value from a static call to method M.

For a given scratch there is typically more than one S, such that *from*[S], or *from**[S] holds⁸. Dealing with this multiplicity of dataflow analysis is natural to DATALOG programming. For example, pattern *dead* identifies dead scratches, i.e., scratches whose values are never used:

```
unassigned := !put[_] ! [ _.from[#] |
  _.func[#] | ...etc. ];
dead := !compared unassigned ;
```

The following predicate selects all sources of values that may be assigned to a scratch:

```
origins[S] := from*[S],
  [ S.parameter | S.constant | S.get[_] ] ;
```

In words, *origins*[S] holds if S is a port of entry of an external value which may eventually be assigned to #. We can now write a pattern which determines whether a method returns a constant.

```
constant_method := method {
  one returned;
  returned => origins: { all constant; }
}
```

Using an overloaded version of *from*[X] which holds if # obtains its value either from a field X or from a call to method X, it is mundane to extend the above pattern to match also method returning a **final** field. It should also be obvious that dataflow analysis provides enough information so that the implementation of pattern capturing, say, a getter notion, or even the selection of fields, is not too difficult.

The native predicate *locus*[S] holds if S and # are distinct scratches which are stored in the same location on the

⁸other than in linear methods, i.e., methods whose control flow contains no branch statements

frame. This predicate is used in Fig. 3.2, together with `dead` and `unassigned` in a bunch of predicates which implement several PMD advices. .

Fig. 3.2 Implementation of some PMD warnings with scratches

```

changed_parameter := parameter locus[_];
only_one_return := method returns: { many; };
null_assignment := from[C], C.null;
unread_parameter := dead parameter;
flag_parameter := unassigned compared parameter;
unassigned_local := local locus: { empty };
unused_local := dead local;

```

4. Underlying Semantics

As stated above, JTL belongs to the logic programming paradigm. In Sec. 4.1 we explain how the JTL constructs are mapped to familiar notions of the paradigm. Sec. 4.2 discusses the (surprising) implications of modelling the task of querying open-ended software (so typical to JAVA) as logic queries over infinite database.

4.1 JTL as a Logic Language

In a nutshell, JTL is a *simply typed* formalism whose underlying semantics is *first order predicate logic* augmented with *transitive closure* (FOPL*). Evaluation in JTL is similar to that of PROLOG (more precisely, DATALOG), with its built-in support for the “join” and “project” operations of relational databases. This section elaborates the language semantics a bit further.

Kinds and Predicates. The type system of JTL consists of a fixed finite set of primitive kinds (types) \mathcal{T} . There are no compound kinds.

A *predicate* is a boolean function of $T_1 \times \dots \times T_n$, $n \geq 0$, where $T_i \in \mathcal{T}$ for $i = 1, \dots, n$. A predicate can also be thought of as a *relation*, i.e., a *subset* of the cartesian product $T_1 \times \dots \times T_n$, called the *domain* of the predicate. By convention, the first argument of a predicate is unnamed, while all other arguments are named. The unnamed arguments is called the *receiver*.

Native Predicates. JTL has a number of *built-in* predicates, such as `class`—a unary predicate of `TYPE`, i.e., `class` \subseteq `TYPE`, `synchronized` \subseteq `MEMBER` (the predicate which holds only for synchronized methods), `members` \subseteq `TYPE` \times `MEMBER`, `extends` \subseteq `TYPE` \times `TYPE` (with the obvious semantics), and the *0-ary* predicates `false` (an empty 0-ary relation) and `true` (a 0-ary relation consisting of a single empty tuple). Built-in predicates are called in certain communities *Extensional DataBase* (EDB) predicates.

A run of the JTL processor starts by loading a declaration of arity and argument types of all native predicates from a configuration file. Native declarations are nothing more than definitions without body. For example, the following commands in a configuration file

```
MEMBER.int;
```

states that `int` is a unary predicate such that `int` \subseteq `MEMBER`.

Compound Predicates. Conjunction, disjunction, implication and negation can be used to define *compound* predicates from the built-ins. Also permitted, are quantification as explained in Sec. 2 and transitive closure, i.e., recursion as in

```

extends+[X] := extends X |
extends[Y] Y.extends+[X];

```

The language offers an extensive library of *pre-defined* compound predicates.

Compound predicates are sometimes called *Intensional Database* (IDB) predicates.

Finite Databases. To run, a JTL program requires a database which *conforms* to the natives, i.e., it must have in its schema the relations or the EDBs as dictated the natives of JTL.

The simplest way to supply a database is by specifying to the JTL processor a finite set of classes and methods, e.g., a “.jar” file. Obviously, such a collection does not directly represent any EDBs. EDBs are realized on top of the collection by means of a bytecode analysis library.

Alternatively, a finite database can also be provided by supplying a finite set of legal JAVA source files. The native relations are then realized on top of these by a JAVA parser.

JTL queries can also be run without a fixed input set. Such a situation, which can be thought of a DATALOG query over an infinite database is discussed below in Sec. 4.2.1.

Evaluation Order. Unlike PROLOG, the order of evaluation in JTL is unimportant. The output set of a pattern is the same regardless of the order by which its constituent predicates in it are invoked. Predicates have no side-effects, and all computations (on finite databases) terminate.

The simplest way to compute the output set is bottom-up, i.e., by using a work-list algorithm which uses the program rules to compute all tuples in all IDBs used by the program goal. This process, although guaranteed to terminate, can be very time- and space-inefficient. JTL instead analyzes queries and applies, whenever possible, a more efficient top-down evaluation strategy

Overloading and Kind Inference. The JTL processor includes a *kind inference engine* which, based on the kind of arguments and arity of the native predicates, infers arity and arguments kinds of predicates defined on top of these. For example, the definition

```
real := double | float;
```

implies that `real` \subseteq `MEMBER`.

JAVA’s overloading of keywords carries through to JTL, e.g., since the JAVA keyword `final` can be applied to classes and members, the built-in predicate `final` in JTL is overloaded, denoting two distinct relations: `final1` \subseteq `TYPE` and `final2` \subseteq `MEMBER`. Many native predicates are similarly overloaded; JTL infers overloading of compound predicates. For example, the conjunction of `final` and `public` is overloaded; the conjunction of `final` and `interface` is not.

Receivers. As seen in the last examples, JTL sports an implicit mechanism of applying a predicate to receivers. A predicate may still explicitly refer to its i^{th} receiver as $\#i$. For example, the above definition of `real` could have been written as

```
real := #1.double | #1.float;
```

The notation $\#$ is shorthand for the entire receivers tuple $\langle \#1, \dots, \#n \rangle$; Thus, an alternative definition of `real` is

```
real := #.double | #.float;
```

Named Arguments. The *signature* of a relation is an ordered pair $\langle \mathbf{R}, \mathbf{A} \rangle$, whose first component, $\mathbf{R} = \langle R_1, \dots, R_n \rangle$, defines the types of the receivers ($R_i \in \mathcal{T}$ for $i = 1, \dots, n$), while the second component, $\mathbf{A} = \{ \langle \ell_1, A_1 \rangle, \dots, \langle \ell_m, A_m \rangle \}$, defines the names of the arguments (the labels ℓ_j , $j = 1, \dots, m$, must be distinct) and their types ($A_j \in \mathcal{T}$ for $j = 1, \dots, m$).

A row of a relation is in general a *named tuple*, i.e., a tuple of values, some of which carrying labels, such that the types of these values and the labels they carry match exactly the signature of the predicate.

Predicates are characterized by signature, e.g., the signature of predicate `members` is $\langle \langle \text{TYPE} \rangle, \{ \langle \text{“M”}, \text{MEMBER} \rangle \} \rangle$, while the definition

```
container[C] := C.members[#]
implies
```

```
<<(MEMBER), {"C", TYPE}>>
```

as the signature of `container`. Overloaded predicates have multiple signatures, one for each meaning. For example, the built-in predicate `final` has two signatures, $\langle\langle\text{MEMBER}\rangle, \emptyset\rangle$ and $\langle\langle\text{TYPE}\rangle, \emptyset\rangle$.

Set Expressions. JTL extends logic programming with what we call a *query*, which is a predicate whose evaluation involves the generation of a temporary relation, and then applying various *set expressions* (e.g., quantifiers) to this relation. A query predicate is true if all the set conditions hold for the generated temporary relation.

The predicate defined in Fig. 4.1 tries to check that a class is “classical”, i.e., that it has at least one field, two or more methods, that all methods are public, all fields are private, that there are no static fields or methods, and that the sets of “setters” and “getters” of this class are disjoint.

Fig. 4.1 A JTL predicate for matching “classical class” notion.

```
classical := class members: {
  has field;
  many method;
  no static;
  method => public;
  field => private;
  disjoint setter, getter;
}
```

(The definition in the figure assumes that predicates `setter` and `getter` were previously defined.)

The essence of the example is the *generator* of the temporary relation, written as `members:`. The colon character (`:`) appended to the predicate `members` makes it into a generator. JTL generates the set of all members M , such that `#.members[M]` holds. This set, which can be also thought of as a relation with only one unnamed column, is subjected to the set expressions inside the curly brackets.

Six conditions are applied to this set: the first is an existential quantifier (`has` is synonymous to `exists`) requiring that at least one element in the generated set satisfies the `field` condition, i.e., that the class has at least one field. The second condition similarly requires that `method` holds to two more members. The 3rd condition, as should be obvious, requires that this set does not contain any static members.

The 4th condition is a *set expression* requiring that the predicate `method implies public` holds in this set, i.e., that method members are public. The 5th condition similarly states that the set of `field` members is a subset of the set of `private` members. Finally, the set expression `disjoint setter, getter` requires that the two subsets obtained by applying predicates `setter` and `getter` to the set of class members are disjoint.

JTL does provide syntax for generating temporary relations with more than one unnamed column. This extra generality is hardly ever needed.

4.2 The Infinite Database Model

Now that we have explained how JTL queries can be understood in terms of logic programs we can march on to discussing some less elementary aspects of the correspondence between JTL and logic queries. Specifically, this section begins with arguing (Sec. 4.2.1) that the software world naturally suggests a perspective by which the underlying database is infinite.

Sec. 4.2.2 then explains the distinction between open and closed

queries based on the infinite database perspective. Sec. 4.2.3 expands the theory of searches in object oriented software repositories by explains the notion of input, output, dependent and unbound arguments of a predicate.

4.2.1 Infinite Databases

The more interesting way of supplying a database is by setting a search path (i.e., a `CLASSPATH` in the JAVA lingo) to the library realizing the EDBs. The computation of an output set then starts at the literals that the program uses, and the progresses along the search path by calls to the library. For example, the evaluation may start from a literal specifying a certain class name, and progress in the search path to find all non-native methods in class which **extends** this given class.

Note however that the set of classes which **extends** a given class is not well-defined. In following a different search path, this set might be different. Moreover, if the search path includes an internet search, then the size of the result is unbounded.

To model this situation, we rely on a notion of an infinite, complete, enumerable and well founded database. Informally, this notion means that:

1. The database is *infinite* in the sense that *an examination of the database can never be exhaustive*. (This situation is typical to all languages with runtime loading, including SMALLTALK [32] and C# [42]; even in languages with no runtime loading, queries starting at a program library can never hope to see *all* the code that uses it.)
2. Although the database is infinite, it is *complete* and *enumerable* in the following sense: For any software module α in the database, it is possible to find all modules α' such that α “uses” α' (for an appropriate meaning of the term “uses”). Thus we assume that all such α' exist in the database (and hence it is complete), and that they can be found, i.e., that the database is enumerable.
3. In saying that the database is *well-founded* we mean that the set of all software modules which are used directly or indirectly by any given module is finite. Therefore, there is no infinite sequence of distinct modules $\alpha_1, \alpha_2, \dots$ such that α_i uses α_{i+1} for all $i = 1, 2, \dots$

Since the general situation that a JTL program faces is such that it must be evaluated against an infinite database, programs hence are classified into two kinds: closed and open. A program is *closed* if (with the above assumptions) it returns a finite set of answers. For example, a program seeking the set of all ancestors of a given class is closed.

A program is *open* if the set of results may be infinite, or in other words, the size of the result may increase indefinitely with the size of finite approximation to the infinite database. The program computing the set of all methods overriding a given method is open.

JTL’s more general semantics is that of queries over infinite databases. Thus, the native predicates may be infinite, e.g., predicate `class` holds for infinitely many values of kind `TYPE` items, and predicate `extends` holds for infinitely many pairs of such values.

Although there is some prior work on the theory of logical queries (see e.g., [65]), the setting described by requirements 2 and 3 above did not receive the attention of researchers. One possible formalization of the above requirements in JTL terms is as follows.

DEFINITION 1. A set of native predicates over an infinite database is converging if all the following conditions hold:

1. All natives are either unary or binary.
2. For every binary predicate p , and for any value ℓ , there are only finitely many values ℓ' such that $p[\ell, \ell']$ holds.
3. There is no infinite chain of distinct values ℓ_1, ℓ_2, \dots such that for all $i = 1, 2, \dots$, there is a native predicate p_i such that $p_i[\ell_i, \ell_{i+1}]$ holds.

Not every infinite database is converging. For example, \mathbb{Z} , the set of integers, with the successor and predecessor relations is not converging. Even the set \mathbb{N} of positive integers with the successor relation is not converging.

As it turns out, and not as a matter of coincidence, the set of native predicates used for processing a programming language is converging; the reason being that if native predicates represent some kind of a “using” relation between modules, then an infinite chain as in Def. 1 implies that there are software modules whose compilation requires a library of infinite size.

4.2.2 Queries in an Open Environment

An interesting challenge in the design of JTL was in dealing with the open world nature of JAVA programs, which is modelled by the infinite database notion described above. Even a simple “open” program such as `int`, generating all `int` typed fields and methods in all classes in the “domain”, carries the flaw that the “domain” is not a well defined notion. Open queries tend to be slower than close queries, since the size of their result increases indefinitely with the finite approximation to the database (roughly speaking, they return larger results in larger finite databases).

Conversely, the output of a closed program only depends on the set of classes transitively used by the classes and methods mentioned in the program.

As part of JTL development, we developed an algorithm that determines whether a given DATALOG query (and hence a JTL program) is open or closed with respect to a converging set of natives.

The description and the correctness proof of this algorithm are beyond the scope of the broad perspective offered by this overview of JTL. However, it should be clear that bottom-up evaluation is not feasible for infinite databases. The algorithm must therefore consider top-down evaluation strategies of queries. As part of its working, the algorithm also generates for all closed programs, a top-down evaluation strategy in which infinite predicates are never exhaustively searched. Instead, the evaluation strategy uses as a basic step queries of the sort described by the second requirement of Def. 1. Convergence is guaranteed by the third requirement in this definition.

Viewed differently, the algorithm can determine whether any static analysis algorithm, if formulated as a logic programming query, makes a whole-world analysis assumption.

DATALOG like queries on an infinite converging database push the expressive power of DATALOG close to its limit. DATALOG semantics on general infinite database falls into the Gödel realm, borders with the Turing-completeness of PROLOG and hence may be too strong. On finite databases, DATALOG is always decidable, which maybe a bit interesting. Converging databases are such that there is an algorithm which determines which queries are decidable.

4.2.3 Input and Output Arguments

PROLOG programmers are constantly cognizant of the fact that due to the recursive and undecidable nature of PROLOG programs, certain arguments of a predicate cannot be computed from others. For example, it is straightforward to set the rules for a PROLOG predicate `mult` (a, b, c) which holds for (large) integers a, b , and c

if $ab = c$. However, although in principle such a predicate can compute all factorizations of a given integer c , it is not likely to do so in reasonable time for a large such integer (with say 1024 bits).

Moreover, given the value a , the predicate should, but cannot return all pairs of b and c such that $ab = c$. PROLOG programmers therefore implicitly (sometimes even explicitly) assign legal “calling patterns” to any predicate they define. In the `mult` example, the calling patterns are that any argument can be computed (or is *output*) given values to the two arguments, i.e., when they serve as *input*.

Calling patterns do not normally exist in the DATALOG world, since all queries can be computed in finite time. However, calling patterns do make sense when DATALOG queries are evaluated over an infinite database.

Another side effect of the algorithm that determines whether a program is open or closed is the computation of calling patterns of all predicates that the program uses. As it turns out, the calling patterns of any predicate defined over a converging database are very simple. Any argument to any predicate, native or compound can be classified into precisely one of the following sorts:

1. *Pure Input*. An argument is pure input if no finite set of values can be computed even from an assignment to all other arguments. For example, the receiver of the binary predicate `#.extends+[M]` is pure input, because its set of legal values cannot be determined from an assignment to the other, named argument.
2. *Pure Output*. An argument is pure output, if a finite superset of all legal assignments to it can be computed even if values of other arguments are not known.

Consider the binary predicate `my_list_suppliers[C]` which holds for all classes C that are used directly or indirectly by the receiver *and* by class `List`. Then, a superset of all legal values of C can be computed regardless of the receiver.

3. *Output*. If an argument is not pure input and not pure output, then there exists a set of determining arguments, such that a finite superset of all legal values to this argument can be computed from *any* assignment to *any* of argument in the determining set.

In the case that the database is not converging (which does not match queries of object oriented programs) it may be the case that an argument can be bounded only after several arguments are bounded.

Consider for example the following quaternary JTL predicate

```
m[A,B,R] := A.declared_in[C] B.declared_in[C]
A.private B.public R.eq[Object];
```

Then, we can see that the receiver is pure input, that A is dependent on B and vice versa, and that R is pure output.

4.2.4 Baggage String Results

The fact that JTL can identify output arguments, and, moreover, it can identify whether an argument is used in determining the value of another, makes it possible to load a “baggage” of output arguments on any native or compound argument.

Such a baggage contributes to the predicate “weight” only if used. If an output parameter is not used by the clients of a predicate, its computation is optimized out at the time of analysis by the JTL program.

JTL offers a special syntax for adding string output parameters to predicates, i.e., parameters of type string. Consider for example the

following JTL definition, taken from our implementation of Eclipse warnings library.

```
method_with_a_constructor_name :=
  method named N declared_in C, C.named[N];
  [* Method # named #name in #C
   looks like a constructor *]
```

The “core” of the predicate is defined between the `:=` and the enclosing `;`. The core matches any method named `N` defined in a class `C`, such that the name of `C` is also `N`. This definition adds a string baggage output parameter, after the enclosing semicolon. This output parameter is written as string literal, which in JTL is wrapped between `[* and *]`.

The baggage output string parameter is unnamed in this case, and can be thought of as an anonymous textual *return value* of the predicate. Such a anonymous baggage return value exists for all native predicates, and is optimized out unless used. For example, the declaration of native predicate `boolean` is

```
MEMBER.boolean; [* boolean *]
```

By default, the return value of conjugated predicate such as `public boolean` is the concatenation of the return value of the conjugated predicates, i.e., the string `public boolean` in this case. The result of a disjunction, defaults to the string result of the disjuncted component which returned the tuple. The string result of negation is obtained by prefixing an exclamation mark to the string result of its argument.

A string literal may refer to other string results. For example, the literal

```
[* Method # named #name in #C
 looks like a constructor *]
```

refers to `#name`, which is the string result of sending the native `name` to `#`. Also, a string literal may obtain the content of variables and parameters. For example, in the above string literal `#` is substituted for a string identification of the method, `#C` for a string identifying its class (as captured in the variable `C`).

JTL allows named string baggage arguments; by convention, a string baggage pure output argument named `Hint` is used by the interpreter to serve a tip to the user. For example, the following is drawn from JTL configuration file, defining the `throws` native.

```
MEMBER.throws[E:TYPE];
Hint [[Method # specifies #E in
 its throws clause.]]
```

(No substitution occurs inside double square brackets enclosing a JTL string literal.)

5. Applications

Having presented the JTL syntax, the language’s capabilities and its underlying semantics, we are in a good position to describe some of the applications.

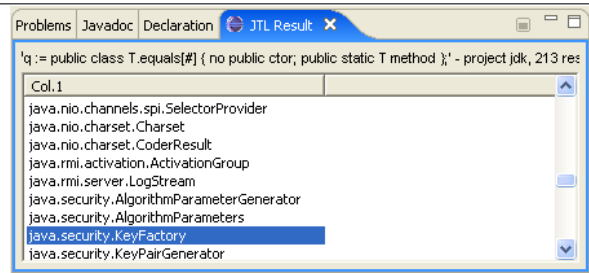
5.1 Integration in CASE Tools and IDEs

In their work on JQuery, Janzen and De Volder [45] make a strong case, including empirical evidence, for the need of a good software query tool as part of the development environment.

As detailed in Sec. 8.1 and demonstrated by Tab. 2, the querying (but not the navigational) side of JQuery can be replaced and simplified by JTL.

We have developed an Eclipse plug-in that runs JTL queries and presents the result in a dedicated view. Fig. 5.1 shows an example: the program (which appears above the results) found all classes in JAVA’s standard library for which instances are obtained using a `static` method rather than a constructor. Using JTL, many searches can be described intuitively. For example, to find

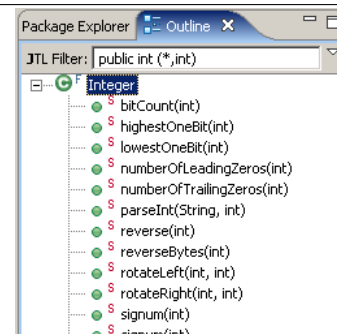
Fig. 5.1 Screenshot of the result view of JTL’s Eclipse plugin



all classes that share a certain annotation `@X`, the developer simply searches for `@X class`. The similarity between JTL syntax and JAVA declarations will allow even developers who are new to JTL to easily and effectively sift through the overwhelming number of classes and class members in the various JAVA libraries.

JTL can also be used to replace the hard-coded filtering mechanism found in many IDEs (e.g., a button for showing only `public` members of a class) with a free-form filter. Fig. 5.2 is a mock screenshot that shows how JTL can be used for filtering in Eclipse.

Fig. 5.2 Using JTL for filtering class members (mock)



Finally, JTL can be used for search-and-replace operations. Since the operation is context-sensitive, there is no risk of, e.g., changing text that appears in comments. With the current version of JTL, this is limited to changing class and method signatures, and is therefore less powerful than Eclipse’s built-in refactoring facilities. However, unlike these facilities, the changes that JTL allows are open-ended and not limited to a pre-defined set of operations. For example, the following JTL program will make every method called `getLock()` synchronized, without changing any other part of the method’s signature:

```
!synchronized getLock()
  [* synchronized #sig *]
```

(`sig` is a library predicate that returns the member’s signature as a valid JAVA source fragment).

5.2 Specifying Pointcuts in AOP

The limited expressive power of the pointcut specification language of ASPECTJ (and other related AOP languages, e.g., CAESAR [54] and ASPECTJ2EE [19]), has been noted several times in the literature [39, 59].

Consider first the following ASPECTJ pointcut specification:

```
call (public void *.set*(*) );
```

JTL’s full regular expressions syntax can be used instead, by first defining

```
setter := void 'set[A-Z]?*' (_); (5.1)
```

and then writing `call(setter)`. Unlike the ASPECTJ version, (5.1) uses a proper regular expression, and therefore does not erroneously match a method whose name is, e.g., `settle()`.

Fig. 5.3 presents an array of ASPECTJ pointcuts trapping read and write operations of primitive public fields. Not only tedious, it is also error prone, since a major part of the code is replicated across all definitions.

Fig. 5.3 An ASPECTJ pointcut definition for all read- and write-access operations of primitive public fields.

```
get(public boolean *) || set(public boolean *) ||
get(public byte *) || set(public byte *) ||
get(public char *) || set(public char *) ||
get(public double *) || set(public double *) ||
get(public float *) || set(public float *) ||
get(public int *) || set(public int *) ||
get(public long *) || set(public long *) ||
get(public short *) || set(public short *);
```

By using disjunction in JTL expressions, the ASPECTJ code from Fig. 5.3 can be greatly simplified if we allow pointcuts to include JTL expressions:

```
primitive := boolean | byte | char | double |
           float | int | long | short;
ppf := public primitive field;
```

```
get(ppf) || set(ppf); // JTL-based AspectJ pointcut
```

The ability to name predicates, specifically `ppf` in the example, makes it possible to turn the actual pointcut definition into a concise, readable statement.

The following is an example of a condition that is impossible to specify in ASPECTJ:

```
setter := public void 'set[A-Z]?*'(_);
boolean_getter = boolean 'is[A-Z]?*'();
other_getter = !boolean !void 'get[A-Z]?*'();
getter := public
    [boolean_getter | other_getter];

field_in_plain_class := public field,
    declare_in[C], C.members: {
        no getter;
        no setter;
    };
```

Condition `field_in_plain_class` holds for `public` fields in a class which has no getters or setters. This requirement is realized by predicate `container`, which captures in `C` the container class. A query is then used to examine the other members of the class.

The above could have been implemented in other extensions of the ASPECTJ pointcut specification language, but not without a loop or a recursive call.

Our contribution puts the expressive power of JTL at the disposal of ASPECTJ and other aspect languages, replacing the sometimes ad-hoc pointcut definition language with JTL's systematic approach.

5.3 Concepts for Generic Programming

In the context of generic programming, a *concept* is a set of constraints which a given set of types must fulfil in order to be used by a generic module. As a simple example, consider the following C++ template:

```
template<typename T>
class ElementPrinter {
public:
    void print(T element) {
        element.print();
    }
};
```

```
}
```

The template assumes that the provided type parameter `T` has a method called `print` which accepts no parameters. Viewing `T` as a single-type *concept* [28, 71], we say that the template presents an implicit assumption regarding the concept it accepts as a parameter. Implicit concepts, however, present many problems, including hurdles for separate compilation, error messages that Stroustrup et al. term “of spectacular length and obscurity” [71], and more.

With Java generics, one would have to define a new interface

```
interface Printable { void print(); };
```

and use it to confine the type parameter. While the concept is now explicit, this approach suffers from two limitations: first, due to the nominal subtyping of JAVA, generic parameters must explicitly implement interface `Printable`; and second, the interface places a “baggage” constraint on the return type of `print`, a constraint which is not required by the generic type.

Using JTL, we can express the concept explicitly and without needless complications, thus:

```
(class | interface) { print(); };
```

There are several advantages for doing that: First, the underlying syntax, semantics and evaluation engine are simple and need not be re-invented. Second, the JTL syntax makes it possible to make useful definitions currently not possible with JAVA standard generics and many of its extensions.

The problem of expressing concepts is more thorny when multiple types are involved. A recent work [28] evaluated genericity support in 6 different programming languages (including JAVA, C# and Eiffel) with respect to a large scale, industrial strength, generic graph algorithm library [68], reaching the conclusion that the lack of proper support for multi-type concepts resulted in awkward designs, poor maintainability, and unnecessary run-time checks.

JTL predicates can be used to express multi-type concepts, and in particular each of the concepts that the authors identified in this graph library.

As an example, consider the `memory_pool` concept, which is part of the challenging example the concepts treatise used by Garcia et al. A memory pool is used when a program needs to use several objects of a certain type, but it is required that the number of instantiated objects will be minimal. In a typical implementation, the memory pool object will maintain a cache of unused instances. When an object is requested from the pool, the pool will return a previously cached instance. Only if the cache is empty, a new object is created by issuing a create request on an appropriate factory object.

More formally, the memory pool concept presented in Fig. 5.4 takes three parameters: `E` (the type of elements which comprise the pool), `F` (the factory type used for the creation of new elements), and `This` (the pool type).

Fig. 5.4 The `memory_pool` concept

```
name create, instance, acquire, release;

factory[E] := (class | interface) {
    public constructor ();
    public E create ();
};

memory_pool[F,E] := equals[T] {
    public static T instance ();
    public E acquire ();
    public release(E);
}, F.factory[E];
```

The body of the concept requires that `This` will provide

acquire() and release() methods for the allocation and deallocation (respectively) of E objects, and a static instance() method to allow client code to gain access to a shared instance of the pool. Finally, it requires (by invoking the Factory predicate) that F provides a constructor with no arguments, and a create() method that returns objects of type E.

As shown by Garcia et al., the requirements presented in Fig. 5.4 have no straightforward representation in JAVA, C# or EIFFEL. In particular, using an **interface** to express a concept presents extraneous limitations, such as imposing a return type on release, and it cannot express other requirements, such as the need for a zero-arguments constructor in a factory. Using an **interface** also limits the applicable types to those that implement it, whereas the concept itself places no such requirement.

In a language where JTL concept specifications are supported, a generic module parameterized by types X, Y and Z can declare, as part of its signature, that X.memory_pool[Y, Z] must hold. This will ensure, at compile-time, that X is a memory pool of Z elements, using a factory of type Y⁹.

Concepts are not limited to templates and generic types. Mixins, too, sometimes have to present requirements to their type parameter. The famous Undo mixin example [4] requires a class that defines two methods, setText and getText, but does not define an undo method. The last requirement is particularly important, since it is used to prevent *accidental overloading*. However, it cannot be expressed using JAVA interfaces. The following JTL predicate clearly expresses the required concept:

```
undo_applicable := class {
    setText(String);
    String getText();
    no undo();
};
```

In summary, we propose that in introducing advanced support of genericity and concepts to JAVA, one shall use the JTL syntax as the underlying language for defining concepts. In addition to the two benefits listed above (simple semantics and evaluation, useful definitions not possible in standard JAVA), using JTL also puts intriguing questions of type theory in the familiar domain of logic, since, as mentioned earlier, JTL is based on FOPL*. For example, the question of one concept being contained in another can be thought of as logical implication. Using text book results [13], one can better understand the tradeoff between language expressiveness and computability or decidability. We are currently working on defining a JTL sub-language, restricting the use of quantifiers, which assures decidability of concept containment.

5.4 Micro Patterns

A μ -pattern is just like a design pattern, except that it is mechanically recognizable. Previous work on the topic [30] presented a catalog of 27 such patterns; the empirical work (showing that these patterns matched about 3 out of 4 classes) was carried out with a custom recognizer. To evaluate JTL, we used it to re-code each of the patterns.

Fig. 5.5 shows the JTL encoding of the Trait pattern (somewhat similar to the now emerging traits OO construct [66]). In a nutshell, a trait is a base class which provides pre-made behavior to its heirs, but has no state.

The code in Fig. 5.5 should make the details of the pattern obvious: A trait is an abstract class with no instance fields, at least one abstract method, and at least one public concrete instance method which was not inherited from Object.

⁹Thus, concepts may be regarded as the generic-programming equivalence of the *Design by Contract* [53] philosophy

Fig. 5.5 The Trait μ -pattern

```
trait := abstract {
    no instance field;
    abstract method;
    public concrete instance method
        !declared_in[Object];
}
```

A programming language researcher could type in the pattern of Fig. 5.5 to quickly find out how many classes in a certain program base are candidates to be implemented as traits.

This example also demonstrates how queries simplify the code. The equivalent PROLOG predicate would have required three recursive calls, probably with the use of auxiliary predicates to implement the three quantifiers in the query.

All patterns were similarly implemented; the specification was never longer than 10 lines. In the course of doing so, we were able to quickly detect ambiguities in the initial textual definition, and check the correctness of the ad-hoc recognizers.

5.5 LINT-like tests.

JTL can be used to express, and hence detect, many undesired programming constructs and habits. On the one hand, JTL's limitation with regard to the inspection of method bodies implies that it cannot detect everything that existing tools [26, 44, 64] can. In its current state, as discussed in Sec. 3.1, JTL cannot detect constructs such as

```
if (C) return true else return false;
```

nor can it easily express numeric limitations (e.g., detecting classes with more than k methods for some constant k).

Yet on the other hand, JTL's expressiveness makes it easy for developers and project managers to improvise and quickly define new rules that are both enforceable and highly self-documenting.

To test this prospect, we a collection of JTL patterns that implement the entire set of warnings issued by Eclipse and PMD (a popular open source LINT tool for JAVA). The only exceptions were those warnings that directly rely on the program source code (e.g., unused **import** statements), as these violations are not represented in the binary class file, that we used.

For example, consider the PMD rule LOOSECOUPLING. It detects cases where the concrete collection types (e.g., ArrayList or Vector) are used instead of the abstract interfaces (such as List) for declaring fields, method parameters, or method return values—in violation of the library designers' recommendations. This rule is expressed as a 45-lines JAVA class, and includes a hard-coded (yet partial) list of the implementation classes. PMD does make a heroic effort, but it will mistakenly report (e.g.) fields of type Vector for some alien class Vector which is not a collection, and was declared outside of the java.util package. The JTL equivalent is:

```
loose_coupling := (class|interface) {
    T method | T field | method(*, T, *);
}, T implements /java.util.Collection;
```

It is shorter, more precise, and will detect improper uses of any class that implements any standard collection interface, without providing an explicit list.

5.6 Additional Applications

Several other potential uses for JTL include encapsulation policies and confined types, among others. *Encapsulation policies* were suggested by Scharli et al. [67] as a software construct for defining which services are available to which program modules. Using JTL,

both the selection of services (methods) and the selection of modules (classes) can be more easily expressed.

Confined types [12] are another example in which JTL could be used, provided of course that confinement is represented in a form of annotation. We have not yet investigated the question of checking the imperative restrictions of confined types with JTL.

6. Performance

The JTL implementation is an ongoing project involving a team of several programmers. The main challenge is in providing robust and efficient execution environment that can be easily integrated into JAVA tools.

The current implementation, which is publicly available at <http://www.cs.technion.ac.il/jtl>, is an interpreter supporting the language core, including top-down evaluation. It does not yet include a complete implementation for some of the more advanced features described earlier, such as string baggage values, and some of the type checking is deferred to runtime.

Work is in progress for a JTL compiler which will generate DATALOG output for an industrial-strength DATALOG engine.

The current implementation uses JAVA's standard reflection APIs for inspecting the structure of a class, and the Bytecode Engineering Library (BCEL) for checking the imperative instructions found within methods. The code spans some 150 JAVA classes that make up a JTL parser, an interpreter, and the implementation of the native predicates that are the basis for JTL's standard library.

On top of this infrastructure there is a text based interactive environment that allows the user to query a given jar file, and an Eclipse plugin that significantly enhances Eclipse's standard search capabilities. Naturally, other application can be easily developed by using JTL's programmatic interface (API).

We will now turn to the evaluation of the performance of this implementation. Our test machine had a single 3GHz Pentium 4 processor with 3GB of RAM, running Windows XP. All JAVA programs were compiled and run by Sun's compiler and JVM, version 1.5.0_06.

In the first set of measurements, we compared the time needed for completing the evaluation of two distinct JTL queries, q_1 and q_2 , defined in Fig. 6.1. Each of the two queries was executed over

Fig. 6.1 JTL queries q_1 and q_2 . $C.q_1$ holds if C declares a public static method whose return type is C ; $C.q_2$ holds if one of the super-classes of C is abstract and, in addition, C declares a `toString()` method and an `equals()` method.

```

q1 := eq[T] declares: { public static T (*) ; };
q2 := extends+ : { abstract ; }
    declares : {
        public String toString() ;
        public boolean equals(Object) ;
    } ;

```

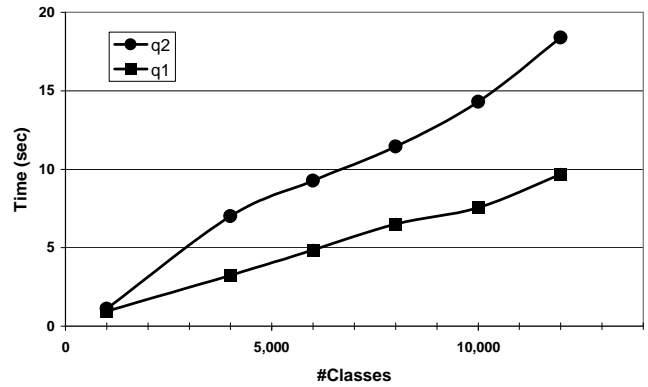
six increasingly larger inputs, formed by selecting at random 1,000, 4,000, 6,000, 8,000, 10,000 and 12,000 classes from the JAVA standard library, version 1.5.0_06, by Sun Microsystems.

The running time of q_1 and q_2 on the various inputs are shown on Fig. 6.2.

Examining the figure, we see that execution time, for the given programs, is linear in the size of the input. The figure may also suggest that runtime is linear in program size, but this conclusion cannot be true in general, since there are programs of constant size whose output is polynomially large in the input size.

The absolute times are also quite reasonable. For example, it took just about 10 seconds to complete the evaluation of program q_1 on

Fig. 6.2 Execution time of a JTL program vs. input size.



an input of 12,000 classes. Overall, the average execution rate for program q_1 was 1,250 classes per second.

In the second set of measurement we compared JTL's Eclipse plugin with that of JQuery. In a similar manner to JTL, JQuery also tries to harness the power of declarative, logical programming to the task of searching in programs, but (unlike JTL) JQuery expressions are written in a PROLOG-like notation.

Another difference between these two systems relates to the evaluation scheme: JQuery uses a bottom-up algorithm for the evaluation of predicates. As explained in Sec. 4, a bottom-up approach is far from being optimal since it needlessly computes tuples and relations even if they cannot be reached from the given input.

Specifically, JQuery initialization stage, where it extracts facts from all classes of the program took more than four minutes on a moderate size project (775 classes), which is two orders of a magnitude slower than JTL's initialization phase. Also the first invocation of an individual JQuery query is roughly ten times slower than the corresponding time in JTL.

Therefore, in order to make the comparison fair to JQuery, we broke a user's interaction with the querying system into a sequence of six distinct stages (defined in Fig. 6.3) and compared the performance of JQuery vs. JTL on a stage-by-stage basis.

Fig. 6.3 The sequence of stages used for benchmarking.

- *Init*. One time initialization
- *Run1*. First execution of the query
- *Run2*. Second execution of the query.
- *Update*. Updating of the internal data-structure following a slight modification of the source files.
- *Run3*. Third execution of the query.
- *Run4*. Fourth execution of the query.

When running the JTL sessions we used the query q_1 defined earlier. In the JQuery sessions we used query q_1' (from Fig. 6.4) which is the JQuery equivalent of q_1 .

Fig. 6.4 The JQuery equivalent of query q_1 . Holds for classes C that declare a public static method whose return type is C .

```

q1' ≜ method(?C, ?M), returns(?M, ?C),
      modifier(?M, static),
      modifier(?M, public)

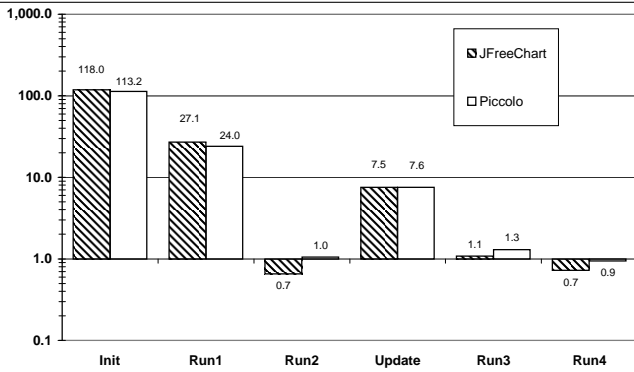
```

We timed the JTL and the JQuery sessions on the Eclipse projects

representing the source of two open-source programs: JFreeChart¹⁰ (775 classes) and Piccolo¹¹ (504 classes). The sizes of these projects, in number of classes, are 775 and 504 (respectively).

The speedup ratio of JTL over JQuery is presented in Fig. 6.5. The figure shows that JTL is faster in the *Init*, *Run1* and *Update*

Fig. 6.5 Speedup of JTL over JQuery, shown on a logarithmic scale. Each pair of columns represents one of the stages defined in Fig. 6.3. Speedup was calculated by dividing the time needed for a stage in the JQuery session with the corresponding time measured from the JTL session.



stages. JTL is about 100 times faster than JQuery at the *Init* stage, and about 25 times faster at the *Run1* stage. JTL was just slightly faster in *Run3*, while JQuery was slightly faster in the *Run2* and *Run4* stages.

As for space efficiency, we predict that a bottom-up evaluator will be less efficient, compared to a top-down evaluator. In particular, we note that running JQuery searches on subject programs larger than 3,000 classes exhausted the memory of the benchmark machine. JTL, on the other hand, was able to process a 12,000-classes project.

7. Beyond Java

As mentioned earlier, JTL has been developed with the Java language in mind. Nonetheless, our theoretical observations, as well as many of the implementation-level concerns, are applicable to other object-oriented programming languages. In this section we will discuss the adaptation of JTL's specification and implementation to other programming languages by considering C[#] as a test case (We will henceforth refer to JTL over C[#] as #TL).

Examining the similarities between JAVA and C[#], we see that in both languages the primary programming construct is the class. A class can define methods and fields, extend another class and implement a number of interfaces. The two languages also agree on the class hierarchy being single rooted, the multiple inheritance in the interface hierarchy, and on the organization of classes in packages (namespaces in C[#] jargon). Many of the keywords of JAVA are also used, with the same semantics, in C[#].

Based on these similarities, JTL can be ported to C[#] simply by replacing the native predicates in JTL's standard library. Some native predicates will have a new implementation (e.g., **class** and **abstract**), a few will be removed (e.g., **transient**), and a few others added (e.g., **const**).

Despite the fact that no other parts of the JTL system need to be changed, this simple port is surprisingly useful. For example, a query such as

```
shared_state := class { no instance field };
```

¹⁰<http://www.jfree.org/jfreechart>

¹¹<http://www.cs.umd.edu/hcil/jazz>

will work on C[#] input if a C[#]-enabled version of the following native predicates is given: **class**, **members**, **static** and **field**. Obviously, most of the non-native predicates in the standard library will also work, with the same semantics, when used with the new native ones.

As in JTL, the implementation of #TL natives may take one of several input formats. In particular, a native predicate may obtain the data it needs from a source code input, or from a compiled binary input (in Microsoft's Intermediate Language, *MSIL*, format).

In order to gain further intuition regarding the porting process, let us consider two C[#] constructs which do not have a JAVA counterpart: **struct** and **delegate**. C[#]'s **structs** are classes which carry value semantics. This means, for example, that instances of **structs** types are stored on the run-time stack, thus delivering some performance gain compared to instances of class types. The natural way to represent **struct** types in JTL would be to add a new native unary predicate, **structs**, that will match values of kind **TYPE** iff they are **structs**. This new predicate is analogous to the **interface** predicate already present in JTL's standard library.

The case of delegates is slightly more complicated. A delegate is a type, parameterized by a method signature, whose values are methods of concrete run-time objects. It may seem that such an entity can only be represented in JTL programs by a new JTL kind, alongside the **TYPE** and **MEMBER** kinds. However, it turns out that JTL is flexible enough to cope with this new language construct by the following two changes, which are (again) limited to the standard library:

First, we will introduce a new native unary predicate, **delegate**, that will match a **TYPE** value *T* iff *T* is a delegate. Second, we will change the implementation of the `call_matching` native predicate. This predicate is used by the JTL interpreter whenever an actual parameters list is matched (as in **static method int compare(T, T)**). Note that this native is unique in that its arity is not fixed, but it is a native predicate nonetheless. Therefore, by changing its implementation such that `call_matching` will match delegate values, and fields of delegate types—in addition to method values—we can make it possible for JTL programs to acknowledge the similarity of delegates and methods.

This discussion is summarized by Fig. 7.1 which presents a query that is written in #TL.

Fig. 7.1 A JTL query that matches C[#] **structs** that define a compare method or a field named `compare` whose type is a two argument delegate.

```
has_compare := struct {
    public int compare(T, T);
};
```

Looking at the body of the `has_compare` predicate (from Fig. 7.1) we see that JTL's systematic semantics remains intact even if the native predicates, making up the standard library, are now examining a program in a different language.

Admittedly, notational differences between JAVA and C[#] may manifest some problems, which will somewhat broaden the abstraction gap between #TL and C[#] (compared to JTL and JAVA).

In particular, the colon symbol, `:`, is used in C[#] to denote inheritance. Therefore, it would have been only natural to use this symbol in #TL to denote the immediate inheritance relationship. In other words, we expect the #TL expression `A : B` to hold if *B* is the direct super-class of *A* or if *B* is a direct super-interface of *A*.

The problem that prevents #TL from supporting such a notation is that the colon symbol already carries a special meaning in #TL (and in JTL): It is the query generator operator. Therefore, if one

wants to turn colon into a predicate, the query generation operator be changed accordingly.

Another issue worth examining is that of dataflow analysis. As described in Sec. 3, JTL's scratch value represent the temporary values that are created during the execution of a JAVA method. It turns out that the execution process of C# is quite similar to that of JAVA; both languages rely on an evaluation stack and on an array of local variables. A typical instruction pops one or more value off the stack and pushes the result back onto it. Finally, the MSIL specification [23] define a verification process that is similar, in principle, to the one defined in the JVM [50] specification

However, despite these similarities, implementing support for **SCRATCH** values on top of the .NET runtime environment is an intricate problem. First, C# allows parameters to be passed by reference (via the **out** modifier), which considerably complicates the data flow analysis algorithm. Second, a C# program may use **unsafe** code blocks. Code in such blocks may break the guarantees which the verifier is trying to establish, thereby reducing the accuracy of the detection of scratches.

We therefore conclude, that implementing support for scratches in #TL is considerably more difficult than in JAVA due to the inherent properties of C#.

8. Discussion and Related Work

Tools and research artifacts which rely on the analysis of program source code are abundant in the software world, including metrics [18] tools, reverse-engineering [8], smart CASE enhancements [43], configuration management [9], architecture discovery [34], requirement tracing [35], AOP [48], software porting and migration [49], program annotation [2], and many more.

The very task of code analysis per se is often peripheral to such products. It is therefore no wonder that many of these gravitate toward the classical and well-established techniques of formal language theory, parsing and compilation [1]. In particular, software is recurrently represented in these tools in an AST.

JTL is different in that it relies of a flat relational model, which, as demonstrated in Sec. 8.2, can also represent an AST. (Curiously, there were recently two works [33, 51] in which relational queries were used in OO software engineering; however, these pertained to program execution trace, rather than to its static structure.)

JTL aspires to be a universal tool for tool writers, with applications such as specification of pointcuts in AOP, the expression of type constraints for generic type parameters, mixin parameters, selection of program elements for refactoring, patterns discovery, and more.

The community has already identified the need for a general-purpose tool or language for processing software. The literature describes a number of such products, ranging from dedicated languages embedded into larger systems to attempts to harness existing languages (such as SQL or XQUERY [11]) to this purpose. Yet, despite the vast amount of research invested in this area, no single industry standard has emerged.

A well-known example is REFINER [63], part of the *Software Refinery Toolset* by Reasoning Systems. With versions for C, FORTRAN, COBOL and ADA, Software Refinery generated an AST from source code and stored them in a database for later searches. The AST was then queried and transformed using the REFINER language, which included syntax-directed pattern matching and compiled into COMMON LISP, with pre- and post-conditions for code transformations. This meta-development tool was used to generate development tools such as compilers, IDEs, tools for detecting violations of coding standards, and more.

Earlier efforts include *Gandalf* [40], which generated a develop-

ment environment based on language specifications provided by the developers. The generated systems were extended using the ARL language, which was tree-oriented for easing AST manipulations. Other systems that generated database information from programs and allowed user-developed tools to query this data included the *C Information Abstractor* [17, 37], where queries were expressed in the INFOVIEW language, and its younger sibling *C++ Information Abstractor* [37], which used the DATASHARE language.

A common theme of all of these, and numerous others (including systems such as *GENOA* [22], *TAWK* [38], *Ponder* [7], *AST-Log* [21], *SCRUPLE* [61] and more) is the AST-centered approach. In fact, AST-based tools became so abundant in this field that a recent such product was entitled *YAAB*, for "Yet Another AST Browser" [6]. Another category of products is contains those which rely on a relational model. For example, the *Rigi* [55] reverse engineering tool, which translates a program into a stream of triplets, where each triplet associates two program entities with some relation.

Sec. 8.1 compares JTL syntax with other similar products. Sec. 8.2 then says a few words on the comparison of relational- rather than an AST- model, for the task of queering OO languages.

8.1 Using Existing Query Languages

*"Reading a poem in translation is like kissing
your lover through a handkerchief."
H. N. BIALIK (1917)*

Many tools use existing languages for making queries. *YAAB*, for example, uses the Object Constraint Language (OCL) [62] to express queries on the AST; the *Software Life Cycle Support Environment* (SLCSE) [70] is an environment-generating tool where queries are written in SQL; *Rigi*'s triples representation is intended to be further translated into a relational format, which can be queried with languages such as SQL and PROLOG; etc.

A more modern system is XIRC [24], where program meta-data is stored in an XML format, and queries are expressed in XQUERY. The JAVA standard reflection package (as well as other bytecode analyzers, such as BCEL) generate JAVA data structures which can be manipulated directly by the language. JQuery [45] underlying is a PROLOG-based extension of Eclipse that allows the user to make queries. Finally, ALPHA [59] promotes the use of PROLOG queries for expressing pointcuts in AOP. We next compare queries made with some of these languages with the JTL equivalent.

Fig. 8.1 depicts an example (due to the designers of XIRC) of using XQUERY to find Enterprise JavaBeans (EJB) which implement `finalize()`, in violation of the EJB specification.

Fig. 8.1 An XIRC search for EJBs that implement `finalize` (from [24]).

```
subtypes (/class[
  @name="javax.ejb.EnterpriseBean"])
/method[
  @name = "finalize"
  and .//returns/@type = "void"
  and not(.//parameter)
]
```

Fig. 8.1 demonstrates what we call *the abstraction gap*, which occurs when the syntax of the queries is foreign to the queried items. The JTL equivalent

```
class implements ' javax.ejb.EnterpriseBean {
  public void finalize();
};
```

is a bit shorter, and perhaps less foreign to the JAVA programmer.

We next compare JTL syntax with that of JQuery [45], which also relies on Logic programming for making source code queries. Tab. 2 compares the queries used in JQuery case study (extraction of

Task	JQuery	JTL
Finding class “BoardManager”	<code>class (?C, name, BoardManager)</code>	<code>class BoardManager</code>
Finding all “main” methods	<code>method (?M, name, main)</code> <code>method (?M, modifier, [public, static])</code>	<code>public static main (*)</code>
Finding all methods taking a parameter whose type contains the string “image”	<code>method (?M, paramType, ?PT)</code> <code>method (?PT, /image/)</code>	<code>method (*, /*image?*/, *)</code>

Table 2: Rewriting JQuery examples [45] in JTL

the user interface of a chess program) with their JTL counterparts. The table shows that JTL queries are a bit shorter and resemble the code better.

The JTL pattern in the last row in is explained by the following: To find a method in which one of the type of parameters contains a certain word, we do a pattern match on its argument list, allowing any number of arguments before and after the argument we seek. The desired argument type itself is a regular expression.

The ASPECTJ sub-language for pointcut definition, just as the sub-language used in JAM [4] for setting the requirements for the base class of a mixin, exhibit minimal abstraction gap. The challenge that JTL tries to meet is to do achieve this objective with a more general language.

Fig. 8.2 is an example of using JAVA’s reflection APIs to implement a query—here, finding all **public final** methods (in a given class) that return an **int**.

Fig. 8.2 Eliciting public final int methods with the reflection library.

```
public Method[] pufim_reflection(Class c) {
    Vector<Method> v = new Vector<Method>();
    for (Method m : c.getMethods()) {
        int mod = m.getModifiers();
        if (m.getReturnType() == Integer.Type
            && Modifiers.isPublic(mod)
            && Modifiers.isFinal(mod))
            v.add(m);
    }
    return v.toArray(new Method[0]);
}
```

When compared with Fig. 8.1, we can observe three things:

- Fig. 8.2 uses JAVA’s familiar syntax, but this comes at the cost of replacing the declarative syntax in Fig. 8.1 with explicit control flow.
- Despite the use of plain JAVA, Fig. 8.2 manifests an abstraction gap, by which the pattern of matching an entity is very different from the entity itself.
- The code still assumes familiarity with an API; it is unreasonable to expect an interactive user to type in such code.

Again, the JTL equivalent, `public final int (*)`, is concise, avoids complicated control flow, and minimizes the abstraction gap.

We should also note that the *fragility* of a query language is in direct proportion to the extent by which it exposes the structure of the underlying representation. Changes to the queried language (i.e., JAVA in our examples), or deepening the information extracted from it, might dictate a change to the representation, and consequently to existing client code. By relying on many JAVA keywords as part of its syntax, the fragility of JTL is minimal.

There are, however, certain limits to the similarity, the most striking one being the fact that in JTL, an absence of a keyword means that its value is unspecified, whereas in JAVA, the absence of e.g., **static** means that this attribute is off. This is expressed as `!static` in JTL.

Another interesting comparison with JTL is given by considering ALPHA and Gybels and Brichau’s [39] “crosscut” language, since both these languages rely on the logic paradigm. Both languages were designed solely for making pointcut definitions (Gybels and Brichau’s work, just as ours, assumes a static model, while ALPHA allows definitions based on execution history). It is no wonder that both are more expressive in this than the reference ASPECTJ implementation.

Unfortunately, in doing so, both languages *broaden* rather than narrow the abstraction gap of ASPECTJ. This is a result of the strict adherence to the PROLOG syntax, which is very different than that of JAVA. Second, both languages make heavy use of recursive calls, potentially with “cuts”, to implement set operations. Third, both languages are fragile in the sense described above

We argue that even though JTL is not specific to the AO domain, it can do a better job at specifying pointcuts. (Admittedly, dynamic execution information is external to our scope.) Beyond the issues just mentioned, by using the fixed point-model of computation rather than backtracking, JTL solves some of the open issues related to the integration of the logic paradigm with OO that Gybels, Brichau, and Wuyts mention [14, Sec. 5.2]: The JTL API supports multiple results and there is no backtracking to deal with.

8.2 AST vs. Relational Model

We believe that the terse expression and the small abstraction gap offered by JTL is due to three factors: (i) the logic programming paradigm, notorious for its brevity, (ii) the effort taken in making the logic programming syntax even more readable in JTL, and (iii) the selection of a relational rather than a tree data model.

We now try to explain better the third factor. Examining the list of tools enumerated early in this section we see that many of these rely on the *abstract syntax tree* metaphor. The reason that ASTs are so popular is that they follow the BNF form used to define languages in which software is written. ASTs proved useful for tasks such as compilation, translation and optimization; they are also attractive for discovering the architecture of structured programs, which are in essence ordered trees.

We next offer several points of comparison between an AST based representation and the the set-based, relational approach represented by JTL and other such tools. Note that as demonstrated in Sec. 8.2, and as Crew’s ASTLog language [21] clearly shows, logic programming does not stand in contradiction with a tree representation.)

1. *Unordered Set Support.* In traditional programming paradigms, the central kind of modules were procedures, which are sequential in nature. In contrast, in JAVA (and other OO languages) a recurring metaphor is the unordered *set*, rather than the *sequence*: A program has a set of packages, and there is no specific ordering in these. Similarly, a package has a set of classes, a class is characterized by a set of attributes and has a set of members, each member in turn has a set of attributes, a method may throw a set of exceptions, etc. Although sets can be supported by a tree structure, i.e., the set of nodes of a certain kind, some programming work is required for set manipulation which is a bit more *natural and intrinsic* to relational

structures.

On the other hand, the list of method arguments is sequential. Although possible with a relational model, ordered lists are not as simple. This is why JTL augments its relational model with built-ins for dealing with lists, as can be seen in e.g., the last row of Tab. 2).

2. *Recursive Structure.* One of the primary advantages of an AST is its support for the recursive structures so typical of structured programming, as manifested e.g., in Nassi-Shneiderman diagrams [57], or simple expression trees.

Similar recursion of program information is less common in modern languages. JAVA does support class nesting (which are represented using the `inners` predicate of JTL) and methods may (but rarely do) include a definition of nested class. Also, a class cannot contain packages, etc.

3. *Representation Granularity.* Even though recursively defined expressions and control statements still make the bodies of OO methods, they are abstracted away by our model.

JTL has native predicates for extracting the parameters of a method, its local variables, and the external variables and methods which it may access, and as shown, even support for dataflow analysis. In contrast, ASTs make it easier to examine the control structure. Also, with suitable AST representation, a LINT-like tool can provide warnings that JTL cannot, e.g., a non-traditional ordering of method modifiers.

It should be said that the importance of analyzing method bodies in OO software is not so great, particularly, since OO methods tend to be small [18], and in contrast with the procedural approach, their structure does not reveal much about software architecture [34]. Also, in the OO world, tools are not so concerned with the algorithmic structure, and architecture is considered to be a graph rather than a tree [43].

4. *Theory of Searches.* Relational algebra, SQL, and DATALOG are only part of the host of familiar database searching theories. In contrast, searches in an AST require the not-so-trivial VISITOR design pattern, or frameworks of factories and delegation objects (as in the Polyglot [58] project). This complexity is accentuated in languages without *multi-methods* or *open classes* [16] but occur even in more elaborate languages. Moreover, some questions of attribute grammars (which are essentially what generates AST) are very difficult, e.g., *EXPTIME*-complete [74].

5. *Data Model Complexity.* An AST is characterized by a variety of kinds of nodes, corresponding to the variety of syntactical elements that a modern programming language offers. A considerable mental effort must be dedicated for understanding the recursive relationships between the different nodes, e.g., which nodes might be found as children or descendants of a given node, what are the possible parent types, etc.

The underlying complexity of the AST prevents a placement of a straightforward interface at the disposal of the user, be it a programmatic interface (API), a text query interface or other. For example, in the *Hammurapi*¹² system, the rule “*Avoid hiding inherited instance fields*” is implemented by more than 30 lines of JAVA code, including two `while` loops and several `if` clauses. The corresponding JTL pattern is so short it can be written in one line:

```
class { field overrides[_] }
```

¹²<http://www.hammurapi.org>

The terse expression is achieved by the uniformity of the relational structure, and the fact that looping constructs are implicit in JTL queries.

*The JTL code in this example is explained as follows: The outer curly parenthesis implicitly loop over all class members, finding all fields among these. The inner ones implicitly loop over all members that this field “overrides”. A match (i.e., a rule violation) is found if the inner loop is not empty, i.e., there **exists** one element in the set for which the boolean condition `true` holds.*

6. *Representation Flexibility.* A statically typed approach (as in Jamoos [31]) can support the reasoning required for tasks such as iteration, lookup and modification of an AST. Such an approach yields a large and complex collection of types of tree nodes. Conversely, in a weakly-typed approach (as in REFINE), the complexity of these issues is manifested directly in the code.

Either way, changes in the requirements of the analysis, when reflected in changes to the kind of information that an AST stores, often require re-implementation of existing code, multiplying the complex reasoning toll. This predicament is intrinsic to the AST structure, since the search algorithm must be prepared to deal with all possible kinds of tree nodes, with a potentially different behavior in different such nodes. Therefore, the introduction of a new kind of node has the potential of affecting all existing code.

In contrast, a relational model is typically widened by adding new relations, without adding to the basic set of simple types. Such changes are not likely to break, or even affect most existing queries.

7. *Caching and Query Optimization.* There is a huge body of solid work on query optimization for relational structures; the research on optimizing tree queries, e.g., XPATH queries, has only begun in recent years. Also, in a tree structure, it is tempting to store summarizing, cached information at internal nodes—a practice which complicates the implementation. In comparison, the well established notion of *views* in database theory saves the manual and confusing work of caching.

9. Further Research

There are several directions in which the work on JTL continues. First, we are currently battling with the recalcitrant issue of extending JTL to also support modifications to the software base. The difficulty here lies with the fact that such changes are expected to preserve the underlying language semantics; in other words, there are complex invariants to which the database under change must adhere. The baggage string results may be used for this purpose. We note however that JTL can be used, as is, for specifying pre- and post-conditions for existing program transformation systems.

Second, we would like to see a type-safe version of embedded JTL, similar to the work on issuing type safe-embedded SQL calls from JAVA [20, 52] and C# LINQ project¹³. The grand challenge is in a seamless integration, a *linguistic symbiosis* [14] of JTL with JAVA, perhaps in a manner similar to by which XML was integrated into the language by Harden *et al.* [41].

Third, it would be interesting to see if the JTL could be enhanced to examine not only the dataflow of methods, but also their control flow. Even more challenging is the combination of the two perspectives.

Fourth, it might be useful to extend JTL to make queries on the program trace, similarly to PQL [51] or PTQL [33]. This extension could perhaps be used for pointcut definitions based on execution stack.

¹³<http://msdn.microsoft.com/vcsharp/future/>

Finally, there is an interesting challenge of finding a generic tool for making language type extensions, for implementing e.g., non-null types [25], read-only types [10], and alias annotations [3]. This could be carried out in the manner described in [29], where the type constraints are specified locally, with two closure conditions: first, a recursively defined constraint on all invocable entities, and second, a condition on allowed modification by inheritance.

The difficulty here lies in the fact that the dataflow analysis we presented is a bit remote from the code. Perhaps the grand challenge is the combination of the brevity of expression offered by JTL with the pluggable type systems of Andreae, Markstrum, Millstein and Noble [5].

10. References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] J. E. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanisms. In *ECOOP'04*.
- [3] J. E. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA'02*.
- [4] D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM Trans. on Prog. Lang. Syst.*, 25(5):641–712, 2003.
- [5] C. Andreae, S. Markstrum, T. D. Millstein, and J. Noble. Practical pluggable types for java. submitted, 2005.
- [6] G. Antoniol, M. D. Penta, and E. Merlo. YAAB (Yet Another AST Browser): Using OCL to navigate ASTs. In *IWPC'03*.
- [7] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *ICSE'96*.
- [8] L. A. Barowski and J. H. Cross II. Extraction and use of class dependency information for Java. In *WCRE'02*.
- [9] L. Bendix, A. Dattolo, and F. Vitali. Software configuration management in software and hypermedia engineering: A survey. In *Handbook of Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing, 2001.
- [10] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA'04*.
- [11] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2005.
- [12] B. Bokowski and J. Vitek. Confined types. In *OOPSLA'99*.
- [13] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.
- [14] J. Brichau, K. Gybels, and R. Wuyts. Towards a linguistic symbiosis of an object-oriented and a logic programming language. In *MPOOL'02*.
- [15] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer, 1990.
- [16] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92*.
- [17] Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.
- [18] T. Cohen and J. Gil. Self-calibration of metrics of Java methods. In *TOOLS'00 Pacific*.
- [19] T. Cohen and J. Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In *ECOOP'04*.
- [20] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE'05*.
- [21] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *DSL'97*.
- [22] P. T. Devanbu. GENOA—a customizable, front-end-retargetable source code analysis framework. *ACM Trans. on Soft. Eng. and Methodology*, 8(2):177–212, 1999.
- [23] ECMA International. Common Language Infrastructure (CLI) Partitions I to VI, 3rd edition. Technical report, ECMA, 2005.
- [24] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. XIRC: A kernel for cross-artifact information engineering in software development environments. In *WCRE'04*.
- [25] M. Fähndrich *et al.* Declaring and checking non-null types in an OO lang. In *OOPSLA'03*.
- [26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*.
- [27] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [28] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA'03*.
- [29] J. Gil and Y. Eckel. Statically checkable design level traits. In *ASE'98*.
- [30] J. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA'05*.
- [31] J. Gil and Y. Tsoglin. JAMOOS—a domain-specific language for language processing. *J. Comp. and Inf. Tech.*, 9(4):305–321, 2001.
- [32] A. Goldberg. *Smalltalk-80: The Interactive Prog. Env*. Addison-Wesley, 1984.
- [33] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA'05*.
- [34] I. Gorton and L. Zhu. Tool support for *Just-in-Time* architecture reconstruction and evaluation: An experience report. In *ICSE'05*.
- [35] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *ICRE'94*.
- [36] G. Gottlob, E. Grädel, and H. Veith. Linear time Datalog for branching time logic. In *Logic-Based Artificial Intelligence*. Kluwer, 2000.
- [37] J. E. Grass and Y. Chen. The C++ information abstractor. In *USENIX C++ '90*.
- [38] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC'96*.
- [39] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD'03*.
- [40] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Trans. Softw. Eng.*, 12(12):1117–1127, 1986.
- [41] M. Harren *et al.* XJ: integration of XML processing into Java. In *WWW'04*.
- [42] A. Hejlsberg *et al.* *The C# Prog. Lang*. Addison-Wesley, 2nd ed., 2003.
- [43] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE'05*.
- [44] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.
- [45] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD'03*.
- [46] J. Järvi, J. Willcock, and A. Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *OOPSLA'05*.
- [47] G. Kiczales *et al.* An overview of AspectJ. In *ECOOP'01*.
- [48] G. Kiczales *et al.* Aspect-oriented programming. In *ECOOP'97*.
- [49] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. A. Müller, and J. Mylopoulos. Code migration through transformations. In *CASCON'98*.
- [50] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd ed., 1999.
- [51] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'05*.
- [52] R. A. McClure and I. H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *ICSE'05*.
- [53] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd ed., 1997.
- [54] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD'03*.
- [55] H. A. Müller and K. Klashinsky. Rigi—A system for programming-in-the-large. In *ICSE'88*.
- [56] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic. The emergent structure of development tasks. In *ECOOP'05*.
- [57] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, 1973.
- [58] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *CC'03*.
- [59] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05*.
- [60] J. K. Ousterhout. Tcl: An embeddable command language. In *USENIX Winter'90*.
- [61] S. Paul and A. Prakash. Querying source code using an algebraic query language. In *ICSM'94*.
- [62] Rational Software. *Object Constraint Language Specification*, 1997.
- [63] Reasoning Systems. *REFINE User's Manual*, 1988.
- [64] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *ISSRE'04*.
- [65] Y. Sagiv and M. Y. Vardi. Safety of datalog queries over infinite databases.
- [66] N. Schärli *et al.* Traits: Composable units of behavior. In *ECOOP'03*.
- [67] N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In *ECOOP'04*.
- [68] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. The generic graph component library. In *OOPSLA'99*.
- [69] C. Smith and S. Drossopoulou. Chai: Traits for Java-like languages. In *ECOOP'05*.
- [70] T. Strelch. The Software Life Cycle Support Environment (SLCSE): a computer based framework for developing soft. sys. In *SDE'88*.
- [71] B. Stroustrup and G. D. Reis. Concepts—design choices for template argument checking. ISO/IEC JTC1/SC22/WG21 no. 1536, 2003.
- [72] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In *OOPSLA'99 Workshop*.
- [73] A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.
- [74] P.-C. Wu. On exponential-time completeness of the circularity problem for attribute grammars. *ACM Trans. on Prog. Lang. Syst.*, 26(1):186–190, 2004.
- [75] M. M. Zloof. Query By Example. In *NAA'75*.