Open Unification for Program Query Languages

Johan Brichau
UCL
Louvain-la-Neuve, Belgium
johan.brichau@uclouvain.be

Coen De Roover
VUB
Brussels, Belgium
coen.de.roover@vub.ac.be

Kim Mens
UCL
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Abstract

Logic-based programming languages are increasingly applied as program query languages *which allow developers to reason about the structure and behaviour of programs. To achieve this, the queried programs are reified as logic values such that logic quantification and unification can be used effectively. However, in many cases, standard logic unification is inappropriate for program entities, forcing developers to resort to overly complex queries. In this paper, we argue that such incidental complexity can be reduced significantly by customizing the unification algorithm. We present a practical implementation approach through inter-language reflection and open unification. These techniques are at the core of the logic program query language SOUL, through which we demonstrate custom unification schemes for reasoning over Smalltalk and Java programs. Queries written in this tailored version of SOUL can exploit advanced program matching strategies without increasing the incidental complexity of the queries.*

1. Introduction

The growing amount of program query languages —of which SOUL [18], JQuery [9], CodeQuest [8] and PQL [13] are only some examples— is testament to the significant momentum on the investigation of a program's structure and/or behaviour by means of user-defined queries. Such queries serve the identification of code exhibiting features of interest which range from application-specific coding conventions [14] over refactoring opportunities [17] and design patterns [5] to run-time errors [3, 13].

A large body of these query languages are *logic* program query languages, meaning that they rely on the use of an executable logic to query the program under investigation. The use of a logic programming language to query programs has several well-established advantages [1, 18]. In imperative programming languages, programmers specify exactly *how* the solution to a problem is to be found using step-by-step algorithms. In contrast, logic programming languages allow the problem itself to be specified. The program will find a solution on its own, relying on a specific problem-solving strategy defined by the language. In such an approach, program queries are expressed as logic conditions over the program's parts. These conditions are grouped into reusable logic rules, while the search for solutions is initiated by launching a logic query.

In order for the above-mentioned logic problem-solving strategy to work, the program under investigation needs to be reified as a value that can be manipulated in the logic language. Such a reification enables the natural use of logic quantification and unification to reason over the program parts. Logic unification essentially establishes a pattern-matching scheme over logic values. As a result, it imposes the same matching scheme over reified program parts. Unfortunately, this predefined matching scheme is often not appropriate for matching reified programs. For example, most program query languages reify the program as a parse tree in the form of logic facts. In such an approach, the pattern matching defined by the logic unification defines a purely syntactic matching of program parts. When reasoning about programs, however, the order of variable declarations, for example, is often unimportant and the matching of variables should also consider their scope. Depending on the purpose of the program queries, there are many more matching and unification strategies that can be envisioned or required.

To cope with these requirements, program query languages often tailor the reified representation of the program to their particular needs. The realisation of more complex pattern-matching is left up to the developers which have to quantify manually over the program to achieve the desired result. Such an approach, however, complicates queries and hampers their declarative nature because developers need to define operational queries that implement the required pattern-matching scheme.

In this paper, we argue that logic program query languages can counter this incidental complexity by opening their unification scheme, essentially allowing a developer to (reflectively) implement a custom definition of unification. Once such custom unification definitions are in place, developers can again rely on the problem-solving strategy of the logic language instead of having to implement such a scheme themselves in each query. We will demonstrate this principle through a disciplined use of inter-language reflection [7] as it constitutes a natural implementation to achieve this "open unification" in logic program query languages. In addition, this technique also ensures a causal connection between the reified logic representation and the actual program from which it was derived. The absence of such a causal link inhibits the direct use of query results by other meta-programming systems and requires that the results exhibit sufficient information to reconstitute the actual program element uniquely. This problem is apparent in program query languages that use a set of generated logic facts, for which no causal link is maintained.

The next section starts our exposition by introducing SOUL, which is the program query language used throughout the remainder of this paper. Section 3 discusses standard logic unification and describes its inconveniences for reasoning and matching programs by means of some examples. SOUL's open unification mechanism, described in section 4, enables the introduction of a unification scheme for Smalltalk programs which is covered by section 5. To conclude, section 6 applies the same technique in the implementation of a unification scheme for Java programs.

2. The SOUL logic program query language

The "Smalltalk Open Unification Language" (SOUL) [16] is a logic program query language implemented in —and tightly integrated with— Smalltalk. SOUL programs are a hybrid combination of Prolog and Smalltalk, meaning that a SOUL program comprises Prolog conditions as well as Smalltalk expressions. This also entails that SOUL programs can manipulate any Smalltalk object as a logic value (i.e. as a constant term) and that these values can be exchanged transparently between logic conditions and Smalltalk expressions.

Its hybrid language characteristic is a crucial element in SOUL's design as a program query language. It provides any SOUL program the ability to manipulate any Smalltalk meta-object seamlessly and to invoke Smalltalk's reflective protocol. As a result, queries in SOUL can use the entire Smalltalk meta-object protocol (MOP) to reason about Smalltalk programs. In addition, SOUL is able to reason about Java programs as well, using an interconnection library between Smalltalk and Java. We will focus on reasoning about Java programs using open unification later on. First, we illustrate SOUL and its use as a program query language by means of some examples.

```
1  ?c isClass if
2    ?c isMemberOf: [Smalltalk allClasses].

3  ?root isAncestorOf: ?directSubclass if
4    ?directSubclass isSubClassOf: ?root.

5  ?root isAncestorOf: ?indirectSubclass if
6    ?indirectSubclass isSubClassOf: ?parent,
7    ?root isAncestorOf: ?parent.

8  if [Object] isAncestorOf: [FooBar]
9  if ?superclass isAncestorOf: [FooBar]
10 if [Object] isAncestorOf: ?subclass
11 if ?superclass isAncestorOf: ?subclass
```

Figure 1. Example SOUL rules and queries.

2.1. Example queries

The SOUL programs presented in this paper employ SOUL's symbiotic syntax [4], which closely resembles Smalltalk's keyworded message syntax. An expression such as `?a plus: ?b is: ?c` is a logic condition that imposes the predicate `plus:is:` over the logic variables *?a*, *?b* and *?c*. It states that the value of *?c* must be the sum of the values of *?a* and *?b*. Apart from this particular syntax, SOUL logic programs are evaluated exactly like Prolog programs. For example, we can use the aforementioned predicate in the query `if 2 plus: 3 is: ?result` to calculate the sum of 2 and 3. Evaluation of this query will produce the result 5, bound to variable *?result*.

A more important distinction between SOUL and Prolog lies in SOUL's ability to embed Smalltalk expressions in the logic program. Such Smalltalk expressions are delimited by brackets and can contain logic variables in exactly the same locations as they can contain Smalltalk variables. For example, `[3.4 asInteger]`, `[1]` and `[?x > ?y]` are Smalltalk expressions embedded in a SOUL logic program. They are evaluated as standard Smalltalk, after logic variables have been substituted by the values they are bound to. Smalltalk expressions can be used both as a logic condition and as a logic value (i.e. a logic term). While the former use will result in its evaluation whenever the condition needs to be proven by the inference process, the latter use will result in its evaluation each time it is unified with another logic term. We will discuss the unification of Smalltalk objects later on. For now, it suffices to understand that they are treated as constants.

Figure 1 presents some example SOUL logic rules and queries. The logic rule on lines 1–2 implements the `isClass` predicate, which can be used to query or check all classes in the Smalltalk image. It concludes that a value of *?c* is a Smalltalk class *if* it is a member of the collection of all classes. This collection is obtained through the evaluation of the expression `Smalltalk allClasses`. SOUL's `isMemberOf:` predicate, which is used to 'iterate' over the collection, behaves exactly like its Prolog counterpart (i.e. `member`), except that it can operate on Smalltalk collections as well as logic lists.

Lines 3 through 7 show the `isAncestorOf:` predicate defined by two logic rules. The first rule expresses that a class *?root* is the ancestor of a class *?directSubclass* if the latter is a subclass of the ancestor class. The second rule expresses that a *?root* class is also the ancestor of a subclass of a class it is already the ancestor of. To find all ancestors of a class `FooBar`, one can launch the logic query on line 9 and the logic programming system will present us bindings for the *?superclass* logic variable for which the `isAncestorOf` relation holds. The queries on lines 8 to 11 demonstrate how the same logic rule can be used to verify whether there is an ancestor relation between two classes (line 8), to find all superclasses of a given class (line 9), but also to find all subclasses of a given class (line 10), or even to find all class pairs for which there exists an ancestor relation (line 11).

2.2. Inter-language reflection

We already mentioned that SOUL's hybrid language characteristic is key to its program querying abilities. In summary, because any Smalltalk object can be used as a value in any SOUL program, we can trivially rely on Smalltalk's MOP to reify any Smalltalk program entity into the logic language. The example implementation of the `isClass` predicate in Figure 1 demonstrates exactly this ability and reifies Smalltalk classes into the logic environment. This ability to reflect on Smalltalk programs using SOUL logic programs has been referred to as *inter-language reflection* [7].

In inter-language reflection, and in reflection in general, the reified representation of a program must be causally linked to the program itself. This means that any change made to the reified representation must be reflected in the program itself, and vice-versa. This property is trivially upheld in SOUL for all reified representations because these are actually meta-objects of Smalltalk. The causal connection is especially practical when the results of a logic program query in SOUL need to be used in other metaprogramming tools. This is an important property for a program query language as it is most often used to extract program parts with the specific intent of performing other operations on these parts (e.g. refactoring, browsing, weaving, etc.). A causal link enables the direct use of query results in such operations. Without the causal link, results of a logic query need to be processed in order to retrieve the actual program parts required by the client tool.

3. Source-code reasoning with standard logic unification

Logic-based program queries, such as those presented in the previous section, rely upon a logic problem-solving strategy for their evaluation. One of the essential operations involved in this problem-solving strategy is *unification*. We briefly recap the essence of logic unification and describe how standard logic unification allows for source code reasoning but does not always exhibit the desired behaviour requiring developers to implement workarounds manually.

3.1. Unification of logic terms

Unification is essentially a pairwise matching process between logic terms that does not only establish an equivalence between logic terms but also assigns values to variables. Two logic terms are said to unify if they match –or– if an appropriate set of bindings can be found for the enclosed variables such that they match when each variable's binding substitutes for the same variable in both terms. Table 1 summarizes the default unification scheme as implemented in SOUL. Since unification is a symmetric operation, we only display the lower-left triangle of the table. The numbers in this table refer to the specifications mentioned in the following paragraphs.

The matching process is straightforward in the case of simple terms such as logic constants, i.e.: two logic constants match if they are the same (#1). For example, the logic constants `a` and `a` unify trivially. Similarly, the logic variable `?x` and the logic constant `b` unify if the unification algorithm establishes a binding of `?x` to `b`. If the variable `?x` was already bound, then its value must unify with constant `b`. The same scheme holds for unification of variables with any other value (#2).

In the case of more complex logic terms, such as compound terms (e.g. `car(wheels,engine,seats)`), the unification algorithm is a recursive process. Compound terms unify if they have an equal number of constituents, if their functors unify and if each constituent unifies with its pairwise counterpart (#3). In the example compound term, the constituents are `wheels`, `engine` and `seats` and the functor is `car`. The logic compound terms `car(wheels,engine, seats)` and `car(wheels,engine, seats)` trivially unify and the terms `boat(keel,engine,seats)` and `car(wheels,engine,seats)` obviously do not unify. As a final example, consider the following terms containing some logic variables: `car(?w,engine,?s)` and `car(wheels,?e,seats)`. These terms unify, given that *?w*,*?e* and *?s* bind to `wheels`,`engine` and `seats` respectively.

	constant	object	compound	variable
constant	1			
object	fail	5		
compound	fail	fail	3	
variable	2	2	2	4

Table 1. Standard logic unification in SOUL.

Finally, it is also important to note that two logic variables unify if their respective values unify. If one of the variables is still free, the free variable is bound to the value of the bound variable. If none of the variables are bound, they become synonyms, which means that they will need to be bound to the same value (#4).

3.2. Unification of Smalltalk objects

SOUL employs a modest extension of the above unification scheme to accommodate the unification of Smalltalk objects. We already mentioned that Smalltalk objects are treated as logic constants in SOUL. Therefore, Smalltalk objects in SOUL unify if they are equal (#5). Smalltalk objects define this equality themselves in the implementation of their '=' method. Table 1 summarizes the default unification scheme as implemented in SOUL.

3.3. Source-code reasoning

Although SOUL's default unification scheme often works well for program elements (such as Smalltalk meta-objects), it does suffer from a number of drawbacks which complicate program querying. Identical meta-objects that reify classes, meta-classes, methods, etc. are guaranteed to be equal and can thus rely on the standard matching based on '=' comparison. However, an identity-based equality of meta-objects that reify sub-method level program parts (e.g. statements) leads to an exact syntactic comparison of method bodies. A matching that is based on such an exact syntactic correspondence is often too conservative for comparing program statements. Issues caused by purely syntactic matching include matching of variable references, semantically-equivalent messages and temporary variable lists.

Variable References: The purely syntactic comparison of variable references merely compares variables by name. Such a unification leads to query results where references to *different* variables, but with *identical names* are equated. In order to unify only references to identical variables, the unification needs to take the scope of the variables into account.

The first query in Figure 2 illustrates the impact this mismatch in the unification scheme has on logic program

```
1  if instanceVariable:?instvar inClass: ?class,
2     method: ?method referencesInstVar: ?instvar

3  if instanceVariable:?instvar inClass: ?class,
4     class: ?subclassOrClass inHierarchyOf: ?class,
5     method: ?method inClass: ?subclassOrClass,
6     method: ?method referencesInstVar: ?instvar,
7     method: ?method definesTemps: ?temps,
8     method: ?method hasArgs: ?args,
9     not(?instvar isMemberOf: ?temps),
10    not(?instvar isMemberOf: ?args)
```

Figure 2. Reasoning over variable references.

queries. The query initiates, in a declarative manner, a search for all methods referring to a certain instance variable. However, as variables are compared by name, this query will in reality find all methods referring to variables sharing the same name. This includes methods that refer to completely different variables but which happen to have the same name. To resolve this issue in the unification of variable references, developers need to encode explicitly that a search for variable references must only consider subclasses of the class in which the variable is defined. Additionally, the query must check that the instance variable isn't shadowed by method arguments or temporary variable declarations. This entire process is encoded in the query on lines 3–10 on Figure 2. In comparison with the query on lines 1–2, it implements a much more operational search and must rely on quantification instead of unification for matching program parts.

Semantically Equivalent Messages: Standard Smalltalk classes define a wide variety of *syntactically different* messages that perform exactly the same behaviour. For example, the messages `ifTrue:ifFalse` and `ifFalse:ifTrue:` are semantically equivalent but their reified meta-objects will not match. Therefore, each query that involves the comparison of messages cannot rely on the standard unification and must manually implement the desired matching through quantification over messages. In this quantification, the developer needs to implement a careful mapping such that corresponding arguments are correctly unified.

Temporary Variable Declarations: A final example drawback of pure syntactic comparison is the order of variable declarations, which is generally unimportant. A list of variable declarations can be considered equivalent without considering the order in which the variables are listed. Unfortunately, standard syntactic comparison of temporary variable lists in Smalltalk does consider the order of declaration.

3.4 Unification with compound terms

To deal with many of the above mismatches, method bodies are often reified as compound logic terms that represent the parse trees of the method bodies. For example, this means that the Smalltalk statement `^ x > y` is reified as the compound term `return(send(var(x),>,var(y)))`. Using such a reification, query developers can use unification to extract parts of the method parse trees and implement a custom comparison through quantification over those parts.

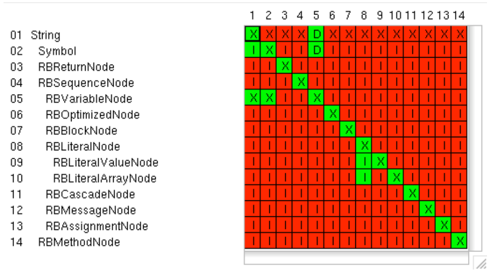
A similar reification of parse trees into logic facts is actually implemented by almost all program query languages. However, it does not provide a general solution to the matching problem. Although it permits query developers to implement custom matching of program parts (instead of relying on the matching defined by unification), it still introduces an additional burden. In essence, developers need to consistently implement the custom matching wherever appropriate and required. An additional problem with parse trees as logic compound terms is that during the traversal, decisions often have to be made by taking the context of the node into account. Maintaining such a context during the traversal introduces additional complications. Finally, the logic compound term representation does not maintain a causal connection with the actual program. All these issues essentially hamper the declarative nature of logic program queries and often exposes developers to intricate details of the reified representation.

In the following section, we present how SOUL overcomes the aforementioned issues by allowing a developer to tailor the unification process to the particular needs of the program querying task at hand.

4. Open unification

SOUL's unification algorithm has been designed and implemented as a customizable implementation, much in the spirit of *open implementations* [11]. Through this open unification mechanism, the developer can customize the unification of Smalltalk objects as well as the unification of any other logic term in the SOUL language. In essence, a custom unification can be defined for each class of objects in the system. This mechanism has allowed us to incorporate more appropriate matching strategies in the unification of reified programs.

Open unification implementation: A specialization of the unification algorithm is achieved by overriding the appropriate methods on the desired class. This is because the SOUL evaluator sends messages to the objects that need to be unified. The implementation of SOUL already defines the default scheme (using '=' comparison) on the `Object` superclass, which allows developers to customize the implementation through overriding. Furthermore, the same messages are also implemented by the Smalltalk class that represents each logic SOUL value. Because Smalltalk allows one to dynamically extend the implementation of these classes, we can therefore also specialize the unification of Smalltalk objects with any SOUL logic value.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
01 String	🟩 X	X	X	X	🟩 D	X	X	X	X	X	X	X	X	X
02 Symbol	🟩 I	🟩 X	I	I	🟩 D	I	I	I	I	I	I	I	I	I
03 RBReturnNode	I	I	🟩 X	I	I	I	I	I	I	I	I	I	I	I
04 RBSequenceNode	I	I	I	🟩 X	I	I	I	I	I	I	I	I	I	I
05 RBVariableNode	🟩 X	🟩 X	I	I	🟩 X	I	I	I	I	I	I	I	I	I
06 RBOptimizedNode	I	I	I	I	I	🟩 X	I	I	I	I	I	I	I	I
07 RBBlockNode	I	I	I	I	I	I	🟩 X	I	I	I	I	I	I	I
08 RBLiteralNode	I	I	I	I	I	I	I	🟩 X	I	I	I	I	I	I
09 RBLiteralValueNode	I	I	I	I	I	I	I	🟩 I	🟩 X	I	I	I	I	I
10 RBLiteralArrayNode	I	I	I	I	I	I	I	🟩 I	I	🟩 X	I	I	I	I
11 RBCascadeNode	I	I	I	I	I	I	I	I	I	I	🟩 X	I	I	I
12 RBMessageNode	I	I	I	I	I	I	I	I	I	I	I	🟩 X	I	I
13 RBAssignmentNode	I	I	I	I	I	I	I	I	I	I	I	I	🟩 X	I
14 RBMethodNode	I	I	I	I	I	I	I	I	I	I	I	I	I	🟩 X

Figure 3. The Open Unification Editor

The protocol's implementation is essentially a double-dispatch pattern that recursively traverses an object-tree representation of the logic values to be unified. Due to space limitations, we refer the interested reader to the SOUL website [16], where the implementation is explained in detail. Although the entire protocol can be customized manually by an application developer, its implementation requires a number of repetitive steps and, more importantly, the developer himself must ensure a correct and symmetrical implementation of the custom unification scheme. Therefore, the open unification implementation is facilitated by a tool: the Open Unification Editor.

Open unification editor: Using this tool, developers can edit a 'unification matrix' which establishes the pairwise unification between reified program elements and other (standard) logic terms. Figure 3 presents a screenshot of the editor opened on Smalltalk's parse tree classes. The diagonal of the matrix identifies that a unification is defined for all objects of the same class (light gray squares). Other combinations either do not unify (dark gray squares) or they inherit the definition of their superclass. The definition itself is implemented by the developer using Smalltalk code but the tool automatically generates the remaining code for the double-dispatch implementation and ensures the symmetry of the algorithm. This allows the developer to concentrate on the implementation of the unification itself without caring about the intricate double-dispatch details.

5. Unification of Smalltalk code

In this section, we describe the custom unification scheme for reasoning over Smalltalk code, essentially tackling the issues mentioned in section 3.3. We defer the presentation of example queries that rely on such a specialized unification to the subsequent section that provides a more elaborate description of a unification scheme for Java code.

5.1. Parse trees and compound terms

Parse trees represented as logic compound terms are appealing from a declarative perspective, especially for their ability to use unification to extract subparts out of parse tree nodes. However, as we pointed out in section 3.3, standard unification is not appropriate for matching parse trees and such a reification is also prone to violating the causal link. Therefore, SOUL reifies method bodies using object-oriented parse trees and establishes a unification between these parse trees and their corresponding logic compoundterm representation. As a result, the advantages of both representations are combined.

The unification extension: A parse tree node with children $c_1, \ldots, c_n$ unifies with a logic term $f(t_1, \ldots, t_n)$ if and only if the term's functor f unifies with the name of the node's class, its multiplicity n agrees with the number of children and each of the term's arguments t_i unifies with the corresponding parse tree node child c_i.

5.2 Messages

We have also specialized the unification of message-send statements, such that syntactically different, but semantically equivalent, messages are matched by the unification algorithm.

The unification extension: A message-send node m_1 with selector s_1 and argument expressions $a_1, \ldots, a_n$ unifies with a message-send node m_2 with selector s_2 and argument expressions $b_1, \ldots, b_n$ if and only if either of the following conditions is true:

1. s_1 and s_2 are syntactically identical and each of the argument expressions a_i unifies with its corresponding argument expression b_i.

2. s_1 and s_2 are semantically equivalent and each of the argument expressions a_i unifies with its corresponding argument expression b_j, given a mapping $i \rightarrow j$ that is particular to each pair s_1 and s_2.

The Smalltalk implementation of this unification extension uses a dictionary to store and retrieve semantically equivalent messages and the appropriate argument mapping. This dictionary is specified by the developer.

5.3 Variable References

Finally, we take the scope of variables into account during their unification.

The unification extension: A variable reference node v_1 with variable name n_1 unifies with a variable reference node v_2 with variable name n_2 if and only if the lookup of variable names n_1 and n_2 in the corresponding lexical scopes of v_1 and v_2 refers to the same variable d.

The lexical name lookup that is required for the above unification scheme is implemented by walking the parse tree from the variable reference nodes up to a node where the variable with the same name is declared. This can either be a temporary variable declaration, a method argument list definition or an instance/class variable definition.

Historically, our primary motivation for adapting the logic unification scheme of SOUL stems from our recent application of SOUL to reason over Java code. In the next section, we therefore present a more elaborate discussion of the unification scheme that we implemented for Java code that lives in an Eclipse workspace.

6 Unification of Java Code

Inter-language reflection, as introduced in Section 2.2, enables SOUL to quantify over any object that is reachable in the Smalltalk image. Similarly, we have established inter-language reflection between SOUL and Java such that SOUL queries an actual Eclipse workspace using an interoperability library between Smalltalk and Java: JavaConnect [10]. This library allows a Smalltalk application to reference any Java object, via a proxy in Smalltalk, and to send it messages. As a result, logic variables are bound to Smalltalk proxies for the actual Java objects. This way, SOUL can seamlessly quantify over projects in the Eclipse workspace, their build options and the parse trees of their contents.

In each of the solutions to the first query depicted in Figure 4, logic variable *?c* is bound to a proxy for a Java compilation unit that declares more than one type. Each such solution is found by backtracking over consecutive bindings for *?c* —returned by the `isCompilationUnit` predicate— until the second condition ([*?c* types size > 1]) is satisfied. Note that this condition is a Smalltalk expression comprising method invocations in Smalltalk that are sent to the actual Java objects using JavaConnect.

Similar to the problems experienced when unifying Smalltalk code, SOUL's standard unification scheme leaves much to be desired for unifying Java's parse tree nodes. To harness the full potential of a logic language to quantify over parse tree nodes, the remainder of this section explores three extensions to the default unification scheme. The first extension defines a unification between logic compound terms and parse tree nodes, for similar reasons as in unifying Smalltalk programs. Its implementation is moreover entirely based on reflection. The two remaining extensions increase quantification efficacy by incorporating static analysis results into the unification.

```
if ?c isCompilationUnit, [?c types size > 1]
if compilationUnit(packageDeclaration(simpleName(['testapp'])),?,?) isCompilationUnit
if ?c isCompilationUnit, ?c hasPackage: ?p, ?p hasName: ?n, ?n isSimpleName, ?n hasIdentifier: [`testapp']
```

Figure 4. Some queries illustrating quantification over the compilation units in an Eclipse workspace.

6.1 Structural reflection for unification

In the logic paradigm, unification with a compound logic term provides an effective means to constrain the values of a variable. Given a suitable reification of parse tree nodes as compound terms, the quantification over all compilation units declared in the package named `testapp` can be accomplished by the second query in Figure 4. However, we have deliberately chosen not to reify parse tree nodes as compound logic terms, which forces us to expand the unification into a quite elaborate sequence of conjuncted conditions comprising the figure's third query.

6.1.1 The unification extension

In order to reconcile the declarative style of the second query with the reification of the actual parse tree node objects, we adapt SOUL's unification scheme such that parse tree nodes unify with their logic compound term representation. Although this extension is much like its corresponding extension for Smalltalk, the Java grammar's sheer size leaves customizing the unification for each individual `ASTNode` subclass a laborious undertaking. Therefore, a reflection-based unification scheme was implemented by means of the *structural property descriptors* available on the entire `ASTNode` hierarchy. A property descriptor for a particular `ASTNode` object essentially provides meta-level information for each of its children, enabling an automatic mapping of the object onto its logic compound term representation.

These reflective capabilities of the `ASTNode`-hierarchy thereby trivialize the unification between a parse tree node and a logic term. Similar to its corresponding extension for Smalltalk code, a parse tree node with property descriptors $\delta_1, \ldots, \delta_n$ unifies with a logic term $f(t_1, \ldots, t_n)$ if and only if the term's functor f unifies with the name of the node's class, its multiplicity n agrees with the amount of property descriptors and each of the term's arguments t_i unifies with the value of the node's property designated by property descriptor δ_i.

Although one can argue that this customization could as well have been part of a closed unification protocol, it has already been overridden for certain subclasses to exclude unimportant properties from the logic term. Import declarations for instance omit their boolean `static` property which can be accessed more naturally in the logic programming paradigm through an `importDeclarationIsStatic` predicate.

6.1.2 The extension in practice

The following logic query illustrates the custom unification in practice:

```
if ?c isCompilationUnit,
   ?c compilationUnitHasPackage: ?p,
   ?c equals: compilationUnit(?p, [?c imports], ?)
```

It is composed of three separate logic conditions. The first condition establishes bindings of the variable *?c* to `CompilationUnit` instances. Through the predicate `compilationUnitHasPackageDeclaration`, the second condition binds *?p* to the package the compilation unit *?c* is declared in. The third condition demonstrates that compilation unit *?c* unifies with the logic compound term given as second argument to the `equals:` predicate[1]. This argument is a Smalltalk term evaluating to the compilation unit's import declarations obtained through an invocation of the Java `CompilationUnit.imports()` method.

6.2 Semantic analysis for unification

As most object-oriented parsers use a distinct object for otherwise structurally indistinguishable nodes, logic program queries often include conditions overcoming SOUL's stringent identity-based comparison of parse tree nodes. Queries involving method selector comparisons, for instance, always comprise conditions comparing the identifiers those selectors encapsulate.

It is therefore tempting to override the identity-based default unification between `SimpleName` instances to obsolete such recurring conditions. However, method declarations do not account for all potential parent nodes of `SimpleName` instances. Relying on an identifier-based structural comparison of names could therefore lead to incorrect or incomplete results. Variable references whose identifiers agree, can still reference a completely different declaration depending on the scope they reside in. Depending on its compilation unit's imports, a simple name can moreover denote the same type as a qualified name. Without explicit support for these complex resolutions, it is unlikely

[1]The logic fact *?x* `equals:` *?x*. implements the `equals:` predicate which hence serves as a substitute for Prolog's = operator.

```
package p;
public class C {
  private Integer f;
  public C(Integer g) { f = g; }
  public C(p.C other) { f = other.f;}
  public Integer notGettingF(Integer f) { return f; }
  public java.lang.Integer getF() { return f;} }
```

Figure 5. A class with an ad-hoc copy constructor and one actual getter method.

that a user's query resolves to both sound and complete results. Indeed, such resolutions are traditionally performed during a compiler's semantic analysis phase.

6.2.1 The unification extension

Including the results of a semantic analysis in the logic reification of the program under investigation isn't a viable option as few application programmers are familiar with its intricate details. Truly declarative program queries are only concerned with a program's entities, leaving any operational details about their matching to the language's problem solving strategy. We therefore extend SOUL's default unification scheme with a comparison between different types of parse tree nodes based on the entities they denote according to a semantic analysis. Eclipse provides the results of its semantic analysis through a `resolveBinding()` method defined on particular parse tree nodes. This method returns a representation of the named program entity the node resolves to. Fully-resolved representations are available for packages, types, methods and the various sorts of variables.

The actual comparison is straightforward for most kinds of nodes. Types are, for example, deemed equal if they resolve to the same entity. This takes care of the aforementioned difference between fully qualified types and types whose resolution depends on package imports. As the next section will demonstrate, it is also convenient to deem type declarations and type references equal if their resolved entities are the same. The same goes for comparing names with field declarations.

Equality among names is somewhat more involved. Naturally, names are deemed equal if they refer to the same entity according to the semantic analysis which takes scoping rules into account. However, identifier-based comparisons are still attempted for names referring to entities of a different kind. This is for instance necessary in queries identifying methods named after the class they are declared in.

6.2.2 The extension in practice

The following rule expresses what it means for a method declaration *?m* to declare a getter method for one of the variable declaration fragments[2] *?g* of a field declaration *?f* in the class *?c*. Its final conditions state that such a method resides in the same class as the field declaration and contains a statement returning an expression that has to unify with the field declaration fragment's name:

```
1 ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2  ?f isFieldDeclarationInClassDeclaration: ?c,
3  ?f fieldDeclarationHasFragment: ?g,
4  ?g variableDeclarationFragmentHasName: ?name,
5  ?m isMethodDeclarationInClassDeclaration: ?c
6  ?m methodDeclarationHasBody: block(?s),
7  ?s contains: returnStatement(?name)
```

SOUL's default unification procedure would result in queries involving the predicate defined above to fail since any candidate field declaration and return statement simply contain different `SimpleName` instances. A custom unification procedure based on the equality of both instance's identifiers would on the other hand return false positives such as the `notGettingF(Integer)` method depicted in Figure 5. While its return statement's expression agrees identifier-wise with the protected field, its parameter shadows the field's name in reality. Finally, `getF()` is correctly identified as the class's single getter method by the unification procedure that relies on semantic analysis for object comparison. In fact, as this comparison is also specified between names and field declaration fragments, it allows for a more concise rule that omits line 4 and has *?g* substitute for the remaining variable *?name* .

Likewise, the following logic rule relies on the unification between class declarations, simple names and types to succeed. It identifies ad-hoc copy constructors which are often the cause of subtle bugs involving a shallow copy where a deep copy was intended instead. The heuristic it employs marks a method as a possible copy constructor if its selector comprises the name of its declaring class and a single parameter of the type that is its declaring class:

```
?m isPossibleCopyConstructorIn: ?c if
  ?m isMethodDeclarationInClassDeclaration: ?c,
  ?m methodDeclarationHasName: ?c,
  ?m methodDeclarationHasParameters: <?p>,
  ?p singleVariableDeclarationHasType: ?c
```

It successfully identifies the third declaration in Figure 5 as a declaration of an ad-hoc copy constructor. For this, the class declaration bound to *?c* was compared with the `Type` instance `p.C` and with the `SimpleName` instance with identifier `C`.

6.3 Points-to analysis for unification

As we have deliberately equaled the base program's reification to its actual implementation, logic queries identifying

[2]Note that Java allows multiple fields of the same type to be declared in one declaration.

software patterns are limited to conditions over parse tree nodes. By nature, such queries are closely related to one particular implementation variant of the pattern they hope to identify. Other variants require a separate logic rule in order to be recognized as an alternative implementation of the same pattern. Unfortunately, this ad-hoc approach often results in an operational definition of the search for pattern variants rather than the desired declarative specification of its essence.

It is however hard to let a single logic rule suffice for the detection of multiple pattern implementation variants without having this rule refer to information about the pattern's run-time behavior. Such behavioral information could be offered by a reification of a static or dynamic analysis of the base program. Unfortunately, only few application programmers are acquainted with the various ways these analyses represent behavioral information. We therefore prefer to incorporate them into the unification procedure, effectively defining the comparison of parse tree nodes as a comparison of the behavior they give rise to.

6.3.1 The unification extension

Our final extension unifies syntactically differing expressions when their values can coincide at run-time according to a static analysis. While the previous extension primarily relieved users from the meticulous application of name resolution and scoping rules, this extension relieves users from the exhaustive enumeration of a base program's aliasing expressions.

To determine whether two expressions can coincide at run-time, the extended unification procedure requires for each expression the set of all heap objects it might point to during an execution of the program. We obtain such points-to sets through the Spark [12] toolkit of the Soot Java Optimization Framework which implements a conservative, flow-insensitive and context-insensitive points-to analysis.

However, as points-to sets are conservative approximations of the actual heap objects a reference points to at run-time, users should be aware that false positives might be reported under the latter. In previous work [2], we have addressed this issue by having the unification succeed with a partial degree of truth, but its fuzzy resolution procedure is out of this paper's scope.

6.3.2 The extension in practice

Under the extended unification procedure, existing pattern detection rules rarely need to be changed in order to recognize a pattern's implementation variants. Unification demands are interpreted in an intentional manner by taking a program's possible run-time behavior into account. As a result, conditions express constraints over the run-time behavior parse tree nodes give rise to instead of constraints over the literal parse tree nodes themselves.

```
package testapp;
public class SumComponentVisitor  {
  private Integer sum;
  public SumComponentVisitor self() { return this;}
  public Integer getSum() { return this.self().sum;}
  public Integer returnSum() {
    return (Integer) returnArg(sum, 10);}
  public Integer retrieveSum() {
    Integer retrieved = self().returnSum();
    return retrieved; }
  public Object returnArg(Object o, int delay) {
    if(delay == 0) return o;
    else return returnArg(o, delay-1); }
}
```

Figure 6. Getter methods in Java.

While the original logic rule for the getter method still recognizes the `getF()` method from Figure 5, its newfound ability to detect implementation variants under the extended unification procedure is better illustrated by the somewhat artificial getter methods depicted in Figure 6. Behavior-wise, one can for instance consider `returnSum()` a getter method since it returns the value of the `sum` instance variable through an invocation of a method which returns its first argument after it has recursively invoked itself as often as indicated by its second argument. For the sake of completeness, we retake the compact version of the `getsFragment:ofFieldDeclaration:in:` predicate's implementation:

```
?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
 ?f isFieldDeclarationInClassDeclaration: ?c,
 ?f fieldDeclarationHasFragment: ?g,
 ?m isMethodDeclarationInClassDeclaration: ?c
 ?m methodDeclarationHasBody: block(?s),
 ?s contains: returnStatement(?g)
```

Its final condition demands that a return statement in the method's parse tree refers to the field it is protecting. Upon evaluation, it expects the unification of the `VariableDeclarationFragment` instance bound to *?g* with the parse tree node corresponding to the return statement's argument expression to succeed. As this rule demonstrates again, it is convenient to have unification defined between variable declarations and arbitrary expressions. A comparison according to semantic analysis will be attempted first. It succeeds for the aforementioned `getF()` method in which the return statement's expression child node is the name of protected field. This comparison will however fail for the other getter methods which feature a field access of an invocation, a cast of an invocation and the name of a local variable respectively. The comparison that is attempted next will however succeed, as it will determine that these parse tree nodes might evaluate to overlapping sets of objects. The return statement's argument expression is, in other words, allowed to be an arbitrarily complicated expression as long as it possibly aliases the protected field's value at run-time. In a sense, the rule's final condition is interpreted as a constraint on the possible values returned by the return statement.

7. Related Work

To the best of our knowledge, the idea of an *open unification* procedure has not been explored before in any logic-based program query language. *Closed extensions* to the unification procedure are however prominent in fuzzy logic programming where the logic problem solving strategy draws conclusions from rules of which the premises are only partially satisfied. Similarity-based unification schemes, in particular, unify two incompatible logic terms provided they are deemed similar up to a certain degree. Similarity degrees are either provided by users [15] or computed by the problem solving strategy itself. Fury [6], for instance, generalizes the well-known edit distance algorithm to logic terms.

8. Conclusion

Logic-based program querying exhibits some well-established advantages regarding concise and declarative expressiveness. Nevertheless, the standard unification scheme often restrains the developer from exploiting the language's full declarative power in program queries. We have proposed *Open Unification* as a mechanism for a query developer to adapt the standard unification scheme to better fit reasoning over program entities. This mechanism essentially permits to relocate the intricate details involved with comparing program entities to the unification algorithm such that developers can implement more concise and declarative program queries. We have demonstrated how some example unification schemes for Java and Smalltalk code significantly simplify program queries in the SOUL logic program query language.

Acknowledgements

Partially funded by the Interuniversity Attraction Poles Programme of the Belgian Science Policy. Johan Brichau is funded by a "FIRST" post-doc grant of the Région Wallonne and Coen De Roover has a PhD scholarship funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders.

References

[1] J. Cohen and T. J. Hickey. Parsing and compiling using prolog. *Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

[2] C. De Roover, J. Brichau, and T. D'Hondt. Combining fuzzy logic and behavioral similarity for non-strict program validation. In *Proc. of the 8th Symp. on Principles and Practice of Declarative Programming*, pages 15–26, 2006.

[3] C. De Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proc. of the 14th IEEE Int. Conf. on Program Comprehension*, pages 202–211, 2006.

[4] M. D'Hondt, K. Gybels, and V. Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proc. of the 2004 Symp. on Applied computing*, pages 1328–1335, 2004.

[5] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Elsevier Int. Journal on Computer Languages, Systems & Structures*, 30(1-2):21–33, 2004.

[6] D. Gilbert and M. Schroeder. Fury: Fuzzy unification and resolution based on edit distance. In *Proc. of the 1st IEEE Int. Symp. on Bioinformatics and Biomedical Engineering*, pages 330–336, 2000.

[7] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Elesevier Int. Journal on Computer Languages, Systems and Structures*, 32:109–124, 2006.

[8] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, 2006.

[9] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd Int. Conf. on Aspect-oriented software development*, pages 178–187, 2003.

[10] JavaConnect:. http://www.info.ucl.ac.be/~jbrichau/software.html.

[11] G. Kiczales, A. Paepcke, and G. Kiczales. *Open Implementations and Metaobject Protocols*.

[12] O. Lhoták. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, December 2002.

[13] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 365–383, 2005.

[14] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proc. of the 13th Int. Software Engineering and Knowledge Engineering Conf.*, 2001.

[15] M. I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.

[16] SOUL:. http://prog.vub.ac.be/SOUL/.

[17] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. of the 7th European Conf. on Software Maintenance and Reengineering*, pages 91–100, 2003.

[18] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, Belgium, January 2001.