Language Symbiosis through Symbiotic Reflection

Roel Wuyts, Stéphane Ducasse
{roel.wuyts | ducasse}@iam.unibe.ch

Software Composition Group
Institut für Informatik
Universität Bern, Switzerland

Abstract. *Meta-programming* is the act of using one system or language to reason about another one. *Reflection* describes systems that have access to and change a causally connected representation of themselves, hence leading to *self-extensible* systems. In reflective languages, only one language is used, while in meta-programming two languages play a role (the base language and the meta language). In this paper we introduce *symbiotic reflection*, a form of reflection between two languages where both languages play the base and the meta-language role. New in this approach is that symbiotic reflection integrates languages from different paradigms in such a way that they both represent, reason about and act upon each other. We illustrate symbiotic reflection with SOUL, a logic programming language in symbiotic reflection with the object-oriented programming language Smalltalk. We show how SOUL does logic reasoning directly on Smalltalk objects, and how to use this to implement *type snooping* and second order logic programming.

Keywords. Reflection - Meta Programming - Language Symbiosis - Logic Programming - Smalltalk

1 Introduction

In todays rapidly evolving world, development environments need to provide sophisticated tools to inspect, navigate and manipulate software systems. Moreover, developers want design tools that are integrated in their development environment, and expect functionality to keep the design documentation and the implementation consistent. Therefore we integrated a logic programming language called SOUL, in the Smalltalk development environment, and used it as a meta-programming language capable of:

– aiding in program understanding: as logic queries are used to interrogate and match abstract syntax trees (AST) of the software system [20];
– help with forward and reverse engineering: we used the logic programming language to express and extract design information (software architectures, design patterns, UML class diagrams and programming conventions) [9, 21].

Using a declarative programming language to reason about other programs is not new. The well known Lint and its derivatives, for example, use regular expressions as the reasoning engine over source code [6], abstract syntax trees [16] or derived source code information [14]. Other approaches use logic programming languages to do the reasoning [10, 3, 11, 12]. All these approaches are essentially meta-programming approaches: the reasoning languages reason about the base languages, making explicit those properties of the base languages they want to reason about [7]. However, it is not possible to change elements of the base language from those declarative languages. This is typically possible in reflective languages, where the aim is to allow *self-extensible* systems [17, 13]. Languages from all paradigms have integrated reflection in some way or another (for example, 2-LISP and 3-LISP [17], M-Lisp [13], CLOS [7, 1], Smalltalk [5, 15], Java and Prolog [2]). Programs in a reflective language have access to a representation of themselves (*introspection*), and can even have the ability to change this (*intercession*).

In this paper, however, we integrate two languages in such a way that

– both can reason about each other, and
– both can modify each other at runtime.

This we call *symbiotic reflection.* Note that meta-programming techniques are not symbiotic reflective because they do not allow the modification of one another. Reflective languages are, by their own definition, symbiotic reflective with themselves. However, in this paper we focus on symbiotic reflection between languages from different paradigms, establishing a tight integration between both. Throughout the paper we illustrate this with examples from SOUL, a logic programming language in symbiotic reflection with the object-oriented programming language Smalltalk. SOUL programs do not only allow one to do pure logic reasoning, but also to:

1. inspect any kind of Smalltalk object;
2. modify any kind of Smalltalk object ;
3. write logic terms that reason about other logic terms.

The structure of the rest of the paper is as follows. Section 2 analyses a motivating example to show what can be obtained with symbiotic reflection. Section 3 then has a look at different kinds of interpreters, including the symbiotic reflective interpreter. Section 4 discusses implementation techniques for implementing a symbiotic reflective interpreter. Section 5 gives another example of using a reflective interpreter, in this case using SOUL to implement a type snooper and some second-order logic predicates. Section 6 discusses the related work. Section 7 concludes the paper.

2 Introductory Example: Scaffolding Support

Before introducing the details of symbiotic reflection, we want to give a concrete example to show the advantages of symbiotic reflection between a logic and an object-oriented programming language. Therefore we use SOUL, a logic programming language that is implemented and integrated with the object-oriented programming language Smalltalk. After we have introduced the example, we discuss the different kinds of meta-programming and reflection that were used.

2.1 Scaffolding Support

The example shows how to investigate all messages sent to the contents of a certain variable, and then how to generate methods for all these messages on another class. Hence it implements support for a prototype development approach (as described by scaffolding patterns [19]) where one starts by implementing a first class, and can then use this implementation to generate the skeleton implementation of the class cooperating with the first class.

Sends. First of all we write a simple logic rule *sends* that relates three arguments: *?c*, *?rec* and *?sends*. It enumerates in the logic list *?sends* all the messages sent to some receiver *?rec* in the context of a class *?c*. It uses other rules *class* and *method* to state that the variable *?c* should be a class and that *?m* should be a method of that class. Then it uses the *sendsTo* rule (not shown in the implementation here, as this is only a short example) to enumerate all the sends to the receiver *?rec* in *?sends*[1]:

```
Rule sends(?c, ?rec, ?sends) if
        class(?c),
        method(?c, ?m),
        sendsTo(?m, ?rec, ?sends).
```

We then use this rule to query the Smalltalk system. For example, we can use this rule to find all the messages sent to a variable *x* in the Smalltalk class *Point*[2]:

```
Query sends([Point], variable(x), ?s)
```

However, besides this use of the *sends* rule that gives a list of all the messages sent to *x*, we can also use it to find in the class *SOULVariable* (the Smalltalk class implementing variables in SOUL) all the expressions (variables, message composition, returns...) that invoke the methods *unifyWith:*, and *interprete:*:

```
Query sends([SOULVariable], ?r, <unifyWith:, interprete:>)
```

[1] Some notes on SOUL syntax:

- the keywords **Rule** , **Fact** and **Query** denote logic rules, facts and queries.
- variables start with a question mark.
- <> is the list notation.
- terms between square brackets are *symbiosis terms*. They contain Smalltalk-like expressions that can reference logic variables from the outer scope.

[2] Note that Smalltalk includes similar functionality by default. However, with this predicate it is very easy to indicate a scope for the search, such as in two unrelated classes or hierarchies.

GenerateEmptyMethod. The second rule is called *generateEmptyMethod*, and generates a Smalltalk method in class *?c* with a given name *?name* (and with an empty implementation). The rule uses an auxiliary predicate *methodSource* that relates the name of a method and a string describing a method with that name (and default arguments, if necessary), that has an empty method body. Then we use a *symbiosis term* represented by [ ] to compile the method *?source* into the class *?c*. *Symbiosis terms* contain Smalltalk-like expressions that can reference logic variables from the outer scope, as explained in Section 4.3. The result of the *symbiosis term* is true or false, depending whether the compilation succeeds or not:

```
Rule generateEmptyMethod(?c, ?name) if
    emptyMethodSource(?name, ?source),
    [(?class compile: ?source)  = nil]
```

The following query creates the method called *abs* in the class *TestNumber*:

```
Query generateEmptyMethod([TestNumber], abs)
```

Generating the interface. We can then combine our two rules to generate methods for the Smalltalk class *TestNumber* for all the messages that are sent to the variable *x* in class *Point*:

```
Query sends([Point], variable(x), ?xSends),
      forall(   member(?xSend, ?Sends),
                generateEmptyMethod([TestNumber], ?xSend))
```

2.2 Example Analysis

This example first of all shows the benefits of using a logic programming language as a meta-programming language to reason about a base language:

– logic programming languages have implicit pattern matching capabilities that make them useful when walking an AST to find certain nodes;
– multi-way: clauses in logic programming languages describe relations between their arguments. These relations can be used in different ways, depending on the arguments passed.
– powerful: it is Turing complete. We used it to express and extract design information such as design patterns or UML class diagrams from the source code. For this, the full power of a programming language is needed, and non-Turing complete languages are generally not sufficient [21].

More importantly, it also demonstrates the different kinds of reasoning and reflection that are available in SOUL (and that we attribute to symbiotic reflection, as we see afterwards):

1. *Introspection.* SOUL terms can reason about their own implementation (as is shown in the query where we use *SOULVariable*).
2. Not shown in this example, but later on in Section 5, is the implementation of second-order logic predicates like *findall, forall, one, calls, ...* in SOUL itself.

3. *Symbiotic Introspection*: we also do logic reasoning directly over Smalltalk objects, i.e., on the meta-language itself. In the example we use Smalltalk classes, that are then inspected to get the methods they implement. It is important to note here that these are the Smalltalk objects themselves that are used, and not decoupled representations.
4. *Symbiotic Intercession*: we use the logic programming language to modify code in the implementation language. Thus, not only can we *inspect* Smalltalk objects, we can also *change* them. For example, the *generateEmptyMethod* rule adds methods to a class. Because the class that is passed is the actual Smalltalk class, adding this method immediately updates the base language.

In symbiotic reflection, the meta-language implements the base language and the base language can reason about and act on the meta-language. Thus the languages are at the same time base language and meta-language for each other.

3 (Symbiotic) Reflective Interpreters

In this section we introduce some classic terminology and definitions of meta-programming and reflection. We then have a look at three different interpreters and discuss their differences. In the following sections we then discuss the implementation of a symbiotic interpreter in general, and the particular case of our example language, SOUL.

3.1 Terminology and Definitions

First of all we want to establish some classic terminology. When implementing an interpreter, the language implementing the interpreter is the *meta-programming language* (hereafter called M), and the interpreted language is the *base language* (hereafter called B). The meta-programming language interprets the program that implements the base language. Both the meta-programming language and the base language manipulate certain data. The difference between a non-reflective, a reflective and a symbiotic reflective interpreter lies in the data they manipulate, as we will show in a moment.

We also first define what is meant by *causally connected*, and by a *reflective system*, since we will use these definitions throughout the rest of the discussion:
Definition: causally connected *A computational system is* causally connected *to its domain if the computational system is linked with its domain in such way that, if one of the two changes, this leads to an effect on the other [8].*
Definition: reflective system *A* reflective system *is a causally connected meta system that has as base system itself [8].*

Definition: Reflection. *Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation:* introspection *and* intercession. *Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called* reification. *[1]*

3.2 Interpreters

We now give an overview of *non-reflective interpreters*, classic *reflective interpreters* and *symbiotic reflective interpreters*, and their differences.

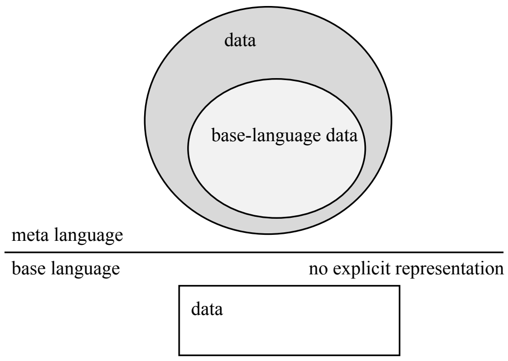


Fig. 1. *A non-reflective interpreter. The base language can only manipulate base level information and not meta-level information.*

Non reflective interpreter. A *non-reflective* interpreter is a program written in the meta-programming language, that needs to evaluate expressions consisting purely of representations of base-language entities. For example, when implementing a Pascal interpreter (in a language L that we leave unspecified here), we will at some point need to implement addition of Pascal numbers. Thus, we will have an L expression to implement this addition, with arguments that represent the Pascal Numbers in L. Hence, from the point of view of the base language (Pascal in the example), there is no interaction with the meta-programming language as shown in Figure 1.

Reflective interpreter. As shown in Figure 2, a reflective interpreter can access and manipulate two kinds of data: (1) the base level data and (2) a causally connected representation of itself, called the *self representation* [17]. During the interpretation the arguments can stem from both levels (but the meta-entities have to be part of the data implementing the base-language). So when interpreting an expression:

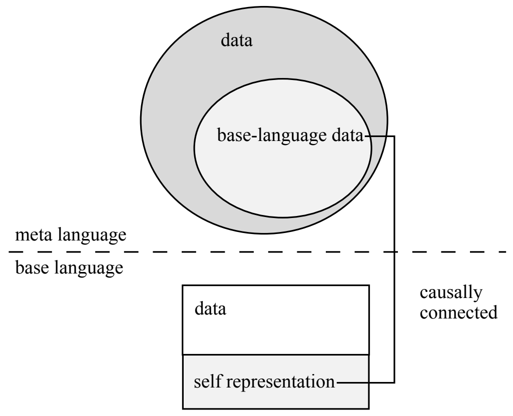


Fig. 2. *A Reflective Interpreter. The base language can access and act on its self-representation*

$interpret(arg_1, arg_2 \dots arg_n)$
the arguments $arg_1, \dots, arg_n$ can be two things:

– base language entities treated at the meta-level,
– self-representing meta-entities.

Symbiotic reflective interpreter. A symbiotic reflective interpreter as shown in Figure 3 is a reflective interpreter that, in addition to being able to manipulate its self-representation can also manipulate the meta-language. As the meta-language implements the base language and the base language can reason about and act on the meta-language, both base language and meta-language can then act and reason on each other.

For example, the following SOUL expression uses the logic term *[Array]*, the logic representation of the Smalltalk class Array:

method([Array], ?m)

The interpreter (a Smalltalk program implementing the SOUL interpretation) manipulates *Array* (a Smalltalk entity that has nothing to do with SOUL's implementation). However, in the following SOUL expression, the interpreter manipulates *?m*, a variable term (a base language entity) and *SOULVariable* (a meta-entity from SOUL's Smalltalk implementation, part of the self-representation):

method([SOULVariable], ?m)

Different meta and base languages. We stress that reflective systems that are written in the same language are in symbiotic reflection because of their uniformity. However, distinguishing symbiotic reflection from reflection is mandatory when different languages are involved where the meta-language can be modified from the base language. The next section shows how to solve the problems that arise during the interpretation of the manipulated entities.

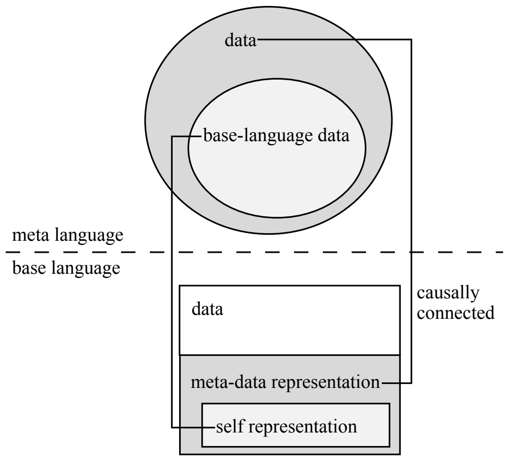


Fig. 3. *A Symbiotic Reflective Interpreter. From the base language it is now possible to access and manipulate the base language self representation and also the meta-level representation*

4 Implementing Symbiotic Reflection between Two Languages

In this section we present problems that occur when the interpreter is symbiotic reflective (i.e., when the base language manipulates its meta-language). Then we show how the upping/downing schema proposes a uniform solution.

4.1 Problems with Handling Objects from Two Different Worlds

Enabling symbiotic reflection between two languages requires that entities of both languages can be manipulated in each language. When the two languages are the same, the task is relatively easy (although not necessarily straightforward) because all the entities share a common data structure. For example, in Smalltalk, the reflective method *instVarAt:* allows one to access the instance variable of any object because it is defined on the class *Object*.

In our case the logic programming language is implemented in the object-oriented programming language, and represents and acts on the object-oriented one. The logic engine is able to manipulate objects as terms and the terms are manipulated as objects. Suppose SOUL would not use the upping/downing schema we present further on, then lots of (implicit or explicit) type checks would be needed to check every time whether we are using a logic term or an object.
A concrete example. In the logic programming language we have a *unify* predicate to unify two arguments. This predicate can be called in different ways: with (representations of) objects and with terms:

Query unify(?c, foo(bar)).

Query unify(?c, [Array]).

The SOUL interpreter somewhere has to implement the unification of two arguments. However, as we see in the logic code, the arguments can be instances of the classes implementing logic terms (like *?c* or *foo(bar)*) as well as objects (like *Array*), that have nothing to do with the implementation of the logic interpreter.

The problem is that the interfaces of these classes differs, since they conceptually belong to two different worlds. The classes implementing logic interpretation typically know how to be unified and interpreted logically, whereas regular objects do not. Possible solutions are:

– All methods in the logic interpretation that come in contact with logic terms need to do *ad-hoc* checks and conversion in the case of a dynamically typed object-oriented programming language or provide several methods with different types in the case of a statically typed object-oriented programming language, or
– implement everything on the root class, so that objects can be used as terms and vice versa. Or,
– convert systematically, but only when needed. Only the 'touching points' between both worlds need to make sure that objects are treated as objects and terms as terms. All the other points are not concerned with the fact that both can exist.

Neither of the first two solutions is satisfactory. In the first one lots of different type-checks have to be done throughout the implementation of the logic interpreter. For the second solution we effectively have to change the implementation language and implement the complete behaviour for the logic interpretation on the root class. The third option is the one we prefer, and that will be discussed in next section.

We would like to stress that such a *transfer* of entities between languages has to be addressed in any language where data structures from the meta-programming language can be manipulated from the base language. At the worse the programmer has to be aware that he is manipulating implementation entities and has to interpret or wrap them himself. This is for example the case in 2-LISP [17], a LISP derivative for studying meta-circularity (as we discuss in Section 6).

4.2 The Upping/Downing Schema

A unified and integrated solution is possible. In our case, it enables objects to be manipulated as logic terms and terms as objects. To explain such a schema we have to introduce two levels: the *up* level and the *down* level.

Symbiotic reflection implies that *both* languages play the base and the meta-language role. The role depends on the view we have on the overall system. From a user point of view, the logic programming language representing and manipulating the object-oriented language acts as a meta language while the object

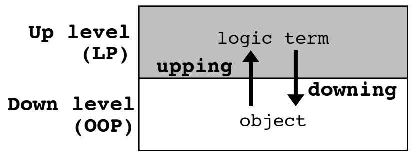


Fig. 4. *The up-down schema allows the uniform manipulation of entities. In our context, it lets Smalltalk objects be directly accessed in SOUL.*

language acts as a base language. However, the logic programming language is implemented in the object-oriented language. So from the interpreter point of view, the object-oriented language is the meta-language and the logic programming is the base language. Hence, it is not clear what we mean by 'meta level' or 'base level' in this context, so from now on we consider *two* conceptual levels as shown in Figure 4.

1. the *down* level is the level of the implementation language of the logic programming language (the object-oriented programming language);
2. the *up* level is the logic programming language level being evaluated by the *down* (object-oriented programming language) level.

Enabling the access and manipulation of down level structures (the object-oriented programming language) from the up level (the logic programming language) in a unified way is possible by following the simple transfer rule: upping a down entity should return an upped entity and downing an upped entity should return a down entity. Applied to SOUL, this rule reads: upping an object should return a term and downing a term should return an object.

This is expressed by the following rules where T represents the set of terms and O the sets of objects, $wrappedAsTerm$ is a function that wraps its argument into a term and $implementationOf$ is a function that returns the data representing its argument.

- (1) $x \in T, x \notin O, up(down(x)) = x$
  For example in SOUL, $up(implementation(?c)) = ?c$
- (2) $x \notin T, x \in O, up(x) = wrappedAsTerm(x)$
  For example in SOUL, $up(1) = [1] = wrappedAsTerm(1)$, where [1] is the logic representation of a term wrapping the integer 1.
- (3) $x \in T, x \notin O, down(x) = implementationOf(x)$
  For example in SOUL, $down(?c) = aVariableTerm$, the smalltalk object representing the logic variable ?c.
- (4) $x \notin T, x \in O, down(up(x)) = x$
  For example in SOUL, $down([1]) = 1$, where [1] is the logic representation of a term wrapping the integer 1.

The transfer rules (1) and (4) are limiting the meta-level to one level. The transfer rule (2) expresses that upping a plain object results in a wrapper that encapsulates the object and acts as a term (and so can be logically unified and interpreted). The transfer rule (3) expresses that downing an ex-nihilo logic term to return the object implementing that term.

4.3 Using the Upping/Downing Schema

Symbiotic reflection requires base symbiotic operators that make the bridge between the base level and the meta-level. These operators are the ones that use the upping and downing to make sure that elements are used 'on the right level' (depending on the operator). They do no need to typecheck, because the up and down schema makes sure that they are at the level that is expected, and wraps or unwraps as needed.

SOUL has one operator that bridges languages, the *symbiosis term*, that relates objects to terms. The *symbiosis term* allows one to use Smalltalk code (parametrized by logic variables) during logic interpretation. It is a logic term that wraps Smalltalk objects and message sends in the logic programming language[3]. From the users point of view the *symbiosis term* takes the form of writing a regular Smalltalk expression that can contain logic variables as receivers of messages, enclosed within square brackets as shown by the examples in Section 2. Conceptually, however, this is not Smalltalk code but a representation of it. However, choosing a familiar representation makes it easy to adopt. For example, LISP uses dotted pairs (lists) to represent structures which, albeit not necessary for enabling the meta-facilities [13], certainly made it easy to adopt them.

When we evaluate a SOUL logic expression to unify terms, we are clearly reasoning at the logic level (the up level). Hence we conceptually think in terms of clauses, terms, unification and backtracking, and expect the result to be a logic result (a logic failure or success, with an updated logic environment containing updated logic bindings). However, the SOUL interpreter is a program in the object-oriented programming language (the down level), so somehow this has to be mapped, taking care that everything is interpreted at the *down level*.

Generally speaking, to interpret an up-level expression:

- we down all elements taking part in that expression;
- we interpret the expression at the down level, and obtain a certain down-level result;
- we up this result.

This solution is analogous to the solutions in for example Lisp, where there is typically a "reduction rule" for quoted pairs, which recursively propagates quotations inwards to the atoms [13]. This recursion is not shown here, but happens also. We now want to give two examples to illustrate this.

[3] For Prolog users: despite its name, a *symbiosis term* can be used both as term and as predication.

Example 1. Let us look at the interpretation of the following SOUL expression (See Section 2):

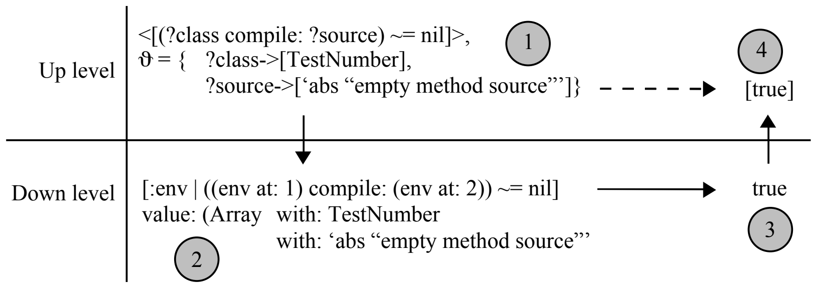
sends([SOULVariable], variable(name), ?xSends)

This expression consists of a compound term, with three arguments. The Smalltalk object representing this logic expression is a parse tree that consists of an instance of class SOULCompoundTerm, that holds on to its arguments. Interpreting the logic expression in a logic context comes down to sending *interprete:* to the parse tree at the Smalltalk level (taking the logic context as an argument). Therefore we down the parse tree and the logic context before sending it *interprete:*. The result of sending *interprete:* is a Smalltalk object, and an updated logic context (containing bindings for the variable *?xSends*). This is then upped to get the logic result.

Example 2. Because of the explicit upping and downing, the evaluation works as well for (representations of) objects as for terms. Let's evaluate the following expression:

[(?class compile: ?source) = nil]

in a logic environment θ where variable *?class* is bound to *[TestNumber]*, and variable *?source* is bound to *'abs "empty method source"'*. This is illustrated in Figure 5.



Up level
<[(?class compile: ?source) ~= nil]>,
ϑ = { ?class->[TestNumber],
?source->['abs "empty method source"']}
1
4
[true]
Down level
[:env | ((env at: 1) compile: (env at: 2)) ~= nil]
value: (Array with: TestNumber
with: 'abs "empty method source"'
true
2
3

Fig. 5. *Interpreting a* symbiosis term *in a logic environment* θ

To interpret the expression, we need to down the expression itself as well as the bindings for the variables in the environment (step 2 in the figure). In this example, the bindings are two Smalltalk objects wrapped as terms (the class *TestNumber* and a string). Downing these terms will yield the objects they contain. The result is the following Smalltalk expression that can be evaluated by the SOUL interpreter:

(TestNumber compile: 'abs "empty method source"') = nil

This piece of Smalltalk code compiles a method in the class *TestNumber*. The source describes a method called *abs*, that contains no statements, but just some comment. The result of sending *compile:* is nil if something went wrong, or the compiled method if it was successful. So, the final result of the complete expression is the Smalltalk object *true* if the method was successfully compiled, and false otherwise (step 3 in the figure). This result is upped to get a result in SOUL: a logic success or a logic failure (step 4).

5 More Examples

In this section we show two examples where SOUL's symbiotic reflection is put to good use: we implement a type snooper and second-order logic predicates. These examples illustrate the transparency in which objects and terms are manipulated in SOUL, due to the symbiotic reflection. In the logic code they behave as logic terms, while inside *symbiosis terms*, they are automatically treated as objects.

5.1 Type Snooper

SOUL comes with a whole framework of rules to reason about the structure of object-oriented (Smalltalk) programs. One of these rules implement a type-inferencer, that not only takes 'classic' type rules into account but also programming conventions and design patterns [21]. To make the advantages of symbiotic reflection clear, we extend the existing type inferencer with a *type snooper*. Type snooping uses the fact that in the Smalltalk development environment objects exist from the class we want to find types of instance variables for. Hence, by looking at these instance variables we find collections of existing types. This is exploited in the rule *objectsForVar* rule shown below. In line 2, the instance variables of the class bound to the variable *?class* are extracted. Then, using the *instVarIndex* predicate in line 3, the index of the variable in the class is bound to *?index*.[4] Lines 4, 5 and 6 then find all the instances of the class for which we want to type the instance variable. From each of these instances we retrieve the object.

```
     Rule objectsForVar(?class, ?var, ?objects) if
1        class(?class),
2        instVar(?class, ?var),
3        instVarIndex(?class, ?var, ?index),
4        generate(  ?objects,
5                   [(?class allInstances collect: [:c |
6                          c instVarAt: ?index]) asStream]).
```

The second rule, *snoopTypeInstVar*, implements the actual type snooping. For an instance variable *?var* of a class *?class* it gives the list of types in the variable *types*. It therefore enumerates all the instances of *class* using the emphobjectsForVar predicate and then, for each of these objects, gets its class (in the symbiosis term on line 9). All these results are enumerated in a list *?allTypes*, from which the doubles are removed to get *types*.

```
     Rule snoopTypeInstVar(?class, ?var, ?types) if
7        findall(   ?cl,
8                   and( objectsForVar(?class, ?var, ?o),
9                        equals(?cl, [?o class])),
```

[4] This is part of the Smalltalk meta-facilities. Objects store their instance variables in slots, that can be accessed with the index. This is done on line 6 where the index is used again.

```
10                     ?allTypes),
11          noDups(?allTypes, ?types).
```

Important in this example is to see the integration between the logic terms and objects. For example, line 3 is a call to a logic predicate that includes three logic variables (that are passed around and unified throughout this rule). But in the symbiosis term on lines 5 and 6 the values bound to the logic variables are treated as objects, without the need to check or convert them. This is possible because SOUL is symbiotic reflective. Without symbiotic reflection integrating such a support is not possible, because we cannot reason about the elements of our base language. In symbiotic reflection we have the objects, and can use them as such, so that we can directly reuse them during the logic interpretation.

5.2 Second-order logic

As a second example we show how to write second-order logic predicates in SOUL. Therefore we reify two concepts that are important during the evaluation of a logic term: the logic repository and the logic environment that holds on to the bindings. We chose to make these two concepts available in the *symbiosis term*, under the form of two hardcoded variables: *?repository* and *?bindings*. The *?repository* variable references the logic repository used when interpreting the *symbiosis term*. The *?bindings* variable holds the current set of bindings. This simple addition makes it possible for a *symbiosis term* to inspect and influence its interpretation. As an example we give the implementation of two widely used logic predicates: *assert* and *one*. The *assert* predicate adds a new logic clause to the current repository. The *one* predicate finds only the first solution of the term passed as argument. If this first solution is found, the bindings are updated and the predicate succeeds, otherwise the predicate fails:

```
Rule assert(?clause) if
      [?repository addClause: ?clause ].

Rule one(?term) if
      [     | solution |
            solution := ( ?term resultStream: ?repository ) next.
            solution isNil
                  ifTrue: [false]
                  ifFalse: [ ?bindings addAll: solution. true ]
      ].
```

We can then use the *assert* predicate to assert facts in the repository, as is done in the following query:

```
Query assert(foo(bar))
```

We want to draw the attention on an important conceptual difference between the examples here when compared with the examples in the type snooper. The symbiosis terms in these examples get passed self-representations, because they manipulate objects that implement the logic clauses and terms. This is clear in the query using assert: the argument is a logic term. In the type snooper, the symbiosis terms were passed 'regular' objects. Because of the symbiotic reflection (and the up/down mechanism used) this does not show up in the symbiosis terms, and no checks are needed.

6 Related Work

One can hardly talk about reflection without discussing the work that has been done in the LISP community, even though the goals of our approach are quite different. One of the very nice features of LISP is that it has a built-in mechanism to represent its language constructs: the quotation form. This provides a core meta-reasoning structure, since parts of programs can be assembled, passed around and then evaluated at will. This basic LISP functionality was extended in the well-known work on *procedural reflection* by Smith [17]. In this work, 2 languages were introduced. The first, 2-LISP, deals with quotation issues by providing two explicit user primitives to switch between representations of structures and the structures themselves. These primitives were called *up* and *down*, a terminology that was used in much of the following work, including this paper. We want to stress two problems with 2-LISP, which is that *up* and *down* needed to be called by the user whenever necessary and that *down* is not the inverse of *up*. 2-LISP was actually meant as the basis for the better-known 3-LISP, a reflective language with an implementation based on *reflective towers*.

In [13] it is written that the quotation form in the original definition of LISP is essentially flawed, and hence that the apply in LISP and descendants like 2-LISP and 3-LISP or even Scheme *crosses levels*. Moreover, it is this level-crossing that allows much of the meta-circular capabilities of LISP. This was remarked by Muller, and was addressed in his LISP flavour, called M-LISP. In M-LISP, the apply function does not cross levels, which removes a lot of the awkward constructions needed in 2-LISP. Moreover, up is represented by a relation R, and down by R^{-1}, where down is the inverse of up. Reification has to be introduced at the cost of equational reasoning (which is done with extended M-LISP), and, even extended M-Lisp corresponds to only a restricted 3-Lisp.

The approach taken by symbiotic reflection is comparable with the approach taken by M-Lisp, except that the goals are quite different. The goal of M-Lisp (as with the other research in reflection in general) is to study 'self-extensibility' of programs, and provide formal language semantics to that end. The approach in our paper is driven from a software engineering problem, to study a symbiosis between two languages from different paradigms. In our case, *up* and *down* therefore not only bridge levels, but also language paradigm boundaries. As in M-Lisp (and contrary to 2-Lisp and 3-Lisp), up and down in SOUL are symmetric and are called automatically. Note however that in SOUL this relation is still kept fairly simple (only objects are reified as logic terms), however this is not necessarily the case. One of the important parts of future work is to have a tighter integration between languages by reifying more object-oriented concepts, which will lead to a more difficult relation.

Besides relating our work to LISP, we also need to mention the work done around Agora [18, 4], a prototype-based language that is symbiotic reflective with its implementation language. The implementation uses an *up/down* mechanism to get reflection with its object-oriented implementation language (Smalltalk, C++ or Java). However, this paper looks at symbiotic reflection between two languages from *different paradigms*.

7 Conclusion

In this paper we presented *symbiotic reflection* as a way to integrate one language (the up-level language) with another language (the down-level language) it reasons about and is implemented in. The benefit is that the up-level language can not only reason about its self-representation (as is the case with classic reflection), but on the complete down-level language. As a result, both languages are meta-language for each other. Symbiotic reflection was illustrated concretely with the object-oriented programming language SOUL, a logic programming language in symbiotic reflection with Smalltalk:

- *Introspection* SOUL terms can do logic reasoning about other SOUL terms;
- *Reflection* SOUL predicates can change data of the SOUL interpreter from within SOUL;
- *Symbiotic Introspection*: SOUL can do logic reasoning about any Smalltalk object;
- *Symbiotic Intercession*: SOUL can do logic reasoning to modify code in the implementation language, impacting the implementation language.

To show the benefits of symbiotic reflection we expressed three non-trivial concrete examples in SOUL. We wrote logic predicates implementing second-order logic operations in SOUL, provided prototype development support and integrated lightweight type-inference with type snooping.

8 Acknowledgments

Thanks to everybody who contributed to this paper: the ex-colleagues from the Programming Technology Lab (where Roel Wuyts did his phd of which this paper is a part) and the people of the Software Composition Group. We want to thank Jacques Malenfant for his comments on an early draft of this paper.

References

1. D. Bobrow, R. Gabriel, and J. White. Clos in context – the shape of the design. In *Object-Oriented Programming : the CLOS perspective*, pages 29–61. MIT Press, 1993.

2. K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-A. Tarnlund, editors, *Logic programming*, volume 16 of *APIC studies in data processing*, pages 153–172. Academic Press, 1982.
3. R. F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.
4. W. De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation. In *Prototype-based Programming*. Springer Verlag, 1998.
5. B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *OOPSLA 89 Proceedings*, pages 327–335, 1989.
6. S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, Dec. 1977.
7. G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
8. P. Maes. *Computational Reflection*. PhD thesis, Dept. of Computer Science, AI-Lab, Vrije Universiteit Brussel, Belgium, 1987.
9. K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.
10. S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, Apr. 1993.
11. N. H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.
12. N. H. Minsky and P. P. Pal. Law-governed regularities in object systems, part 2: A concrete implementation. *Theory and Practice of Object Systems*, 3(2):87–101, 1997.
13. R. Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4):589–616, Oct. 1992.
14. G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.
15. F. Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, February 1996.
16. D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.
17. B. C. Smith. Reflection and semantics in LISP. In P. C. K. Kennedy, editor, *Principles of Programming Languages. Conference Record of 11th Annual ACM Symposium, Salt Lake City, UT, Jan. 15-18, 1984*, number ISBN 0-89791-125-3, pages 23–35, New York, 1984. ACM.
18. P. Steyaert and W. D. Meuter. A marriage of class- and object-based inheritance without unwanted children. In W. Olthoff, editor, *Proceedings ECOOP'95*, LNCS 952, pages 127–144, Aarhus, Denmark, Aug. 1995. Springer-Verlag.
19. J. Vlissides, J. Coplien, and J. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley Publishing Company, 1996.
20. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.
21. R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.