Linguistic Symbiosis through coroutined interpretation

Theo D'Hondt, Kris Gybels, Maja D'Hondt and Adriaan Peeters
Computer Science Department
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{tjdhondt,kgybels,mjdhondt,adpeeter}@vub.ac.be

May 9, 2003

1 Introduction

In the world of computer languages linguistic symbiosis describes the process whereby multiple programming styles or paradigms are instantiated in one composite language. In some cases symbiosis is natural because one particular paradigm can easily be expressed on top of another; in others it is easy to construct because the cohabitation of some paradigms is obvious. Languages like Scheme merge a functional style with an imperative one without too much hassle; in a similar vein C++ merges objects into a procedural language. The object-oriented paradigm seems to be compatible with most others; the concurrent programming paradigm less so; and the logic programming paradigm seems to be most resistant to symbiosis, although logic can be eminently used to formulate other paradigms.

In this day-and-age of domain specific languages, aspect oriented technology, etc. we are confronted by a need to assemble custom languages by combining multiple styles and paradigms, or one of the myriad current or future sub-paradigms. There is no such thing as a universal programming language and technologies such as XML, which are nothing other than (albeit badly designed) variations of Lisp, set the stage for the "reinvention" of language engineering and meta-programming. In an environment where information is formulated in a dedicated language and is annotated with the rules for its interpretation, the notion of linguistic symbiosis emerges as something important. Factoring out paradigms, making them reusable and combinable in concrete instances of custom programming languages seems to be an essential step towards true language engineering.

We are interested in true symbiosis: a context that allows expressions to be formulated in all of the participating paradigms, without preferential treatment of one or the other. We want to be able to write:

```
(let ((total 0)
      (sum (query (x y) (positive-logic-sum x y 10))))
   (for-each*
      (lambda (x y) (set! total (+ total (* x y))))
      (sum x y)))
```

in a fictional Scheme-like procedural-logic symbiotic language in order to compute the sum of products of all pairs of natural numbers adding up to 10. Obviously, syntax is an issue. Developing a language that allows access to two paradigms with equal comfort is far from simple. Making objects explicit in C without traumatizing the odd C-programmer takes some doing and C++ was successful in doing that. However, C++ is an example of an asymmetric solution, forcing the object syntax to comply with the C syntax. This is not necessarily what we want. Moreover, getting the syntax right often detracts from the real and most difficult issue: finding a stable marriage of two, possibly wildly different sets of semantics.

Hence we propose to treat language symbiosis at the syntactic level and at the semantic level as two different activities. In this discussion we will only address the latter, as the most difficult one although not necessarily the most important one. Consider for instance the language Schelog [SCHELOG] which is one of the very few embeddings of a logic language into a procedural one. It suffices to inspect an example:

```
(define %factorial
  (%rel (x y x1 y1)
    [(0 1)]
    [(x y) (%is x1 (- x 1))
           (%factorial x1 y1)
           (%is y (* y1 x))]))
```

to see that the fairly clean semantics of the language can be instantly killed by a less than accessible syntax. Our approach to separating syntax from semantics is to consider a language's interpreter and abstract grammar as one entity, and to consider as a different entity the mapping between concrete and abstract grammar. This is actually more or less the same as separating the denotational semantics from the syntax specification, without requiring us to adopt an approach that is possibly too formal to make our point.

In the following sections we consider minimal interpreters for two fictional and equally minimal languages. One is procedural (hence the P-suffix) and is actually a subset of Pico [PICO]; the other is inspired by the query evaluator in [SICP] and supports a simple logic programming paradigm (hence the L-suffix).

The point that we want to make is the following: imagine we want to combine P and L into a symbiotic LP; imagine furthermore that we want to perform this through the composition of the abstract grammars and interpreters of both languages. Our conjecture is that the establishment of a coroutine relationship between evaluator^P and evaluator^L does the trick. Hence the title.

2 Language P: procedural

The abstract grammar of the language^P can be written as a standard context free grammar with the terminals being a set of basic values and tags. We will write the production rules for the language as follows:

<non-terminal> ::= <tag> <non-terminal>*

The tags are used by the evaluation function for the language to dispatch to an appropriate evaluation subroutine. The evaluation function takes a set of expressions to evaluate and the environment to evaluate them in as arguments and produces a value, its signature is thus:

$$\text{evaluate}^P\text{: expressions}^P \times \text{environment}^P \longrightarrow \text{values}^P \subset \text{expressions}^P$$

Below are the production rules for a simplified version of the language, allowing the definition and assignment of variables, construction of procedures and applying them to arguments:

```
expression  ::= number | text | table | variable |
                application | definition | assignment
number      ::= NBR number
text        ::= TXT text
table       ::= TAB expression*
procedure   ::= PRO parameters expression environment
variable    ::= VAR text
application ::= APL variable table
definition  ::= DEF ( variable | application ) expression
assignment  ::= SET ( variable | application ) expression
parameters  ::= table
environment ::= table
```

Values are expressions that are invariant with respect to evaluation: numbers, text, tables and procedures. Procedures are first class and are stored as closures. Variable definitions and assignments are registered in environments which are lists of variable-value pairs and implement the scope/visibility rules of the language.

3 Language L: Logic

The logic language's evaluation function has the same signature as the one for the P-language:

$$\text{evaluate}^L\text{: expressions}^L \times \text{environment}^L \longrightarrow \text{values}^L \subset \text{expressions}^L$$

Of course the function will evaluate different expressions, which follow these grammar rules:

```
expression  ::= number | text | list | rule | variable |
                pattern | clause
number      ::= NBR number
symbol      ::= SYM text
list        ::= LST expression*
rule        ::= RUL pattern pattern environment
variable    ::= VAR symbol
pattern     ::= PAT symbol terms
definition  ::= DEF pattern pattern
terms       ::= list
environment ::= list
```

Both predicates and functors are captured by a pattern; clauses are defined by associating a head and body pattern; pattern terms are typically values, variables or functors. Note that we provide first class rules, in view of binding them in a P-context. A rule contains an environment which is actually a modular rule-base.

4 Language LxP: the Combination

We now turn to the question of how we can combine language^P and language^L into a new symbiotic language^{LP}:

$$\text{evaluate}^{LP}\text{: expressions}^{LP} \times \text{environment}^{LP} \longrightarrow \text{values}^{LP} \subset \text{expressions}^{LP}$$

We want to combine the abstract grammars of the two languages so that for example a pattern can now contain expression^L's but also expression^P's. The new abstract grammar should thus be the combination of all of the production^P's and all of the production^L's. This implies more than a simple merge however.

A first important issue concerns a suitable structure for environments, as they were almost certainly different in the original languages. We might pursue an implementation of a common environment structure: a chronologically built linked list of variable-value pairs that directly implements the visibility rules. On the other hand, anticipating the coroutine structure of the participating interpreters, we might want to embed each environment in the closure of the interpreter that it applies to.

A second important question is how we can write an interpreter for evaluate^{LP} without having to start from scratch; we obviously want to reuse the interpreters for the original languages. Our proposed answer is to implement evaluate^{LP} by wrapping evaluate^P and evaluate^L inside coroutines and have them communicate as coroutines.

Let us illustrate how such an evaluation would proceed with a simple example of multi-paradigm programming: a route planner using a branch and bound algorithm (taken from [ASPECT]). This search strategy is different from the one usually used in logic languages and while it is possible to implement it in a logic language, a procedural language is better suited. On the other hand, the facts and rules stating the possible routes are easier to implement and manipulate in a logic language. With the LP language it should be possible to implement both parts in languages most suited to each. Figure 4 shows part of the code for this example. Note that we give the example in a fictitious concrete syntax for the sake of clarity even though we are focusing on the interplay between the two interpreters when manipulating the abstract syntax. We specifically leave open the question of whether this particular concrete syntax can be molded into an unambiguously parsable concrete grammar.

```
1   link("Haacht",     "Keerbergen",    5);
2   link("Haacht",     "Boortmeerbeek", 6);
3   link("Haacht",     "Rijmenam",      7);
4   link("Bonheiden",  "Boortmeerbeek", 7);
5   link("Bonheiden",  "Keerbergen",    9);
6   votes("Haacht", ...
7   votes("Keerbergen", ...
8   votes("Bonheiden", ...
9   votes("Boortmeerbeek", ...
10  votes("Rijmenam", ...
11
12  prohibited("Rijmenam", "Bonheiden");
13
14  isfree(?node) :-
15    not(visited(?node)).
16
17  prohibited(?from, ?to) :-
18    link(?from, ?to, ?dist),
19    greater(percentage_votes(?to, "VLB"), 20)
20
21  percentage_votes(node, party): {
22    votes(node, party, ?votes);
23    total := 0;
24    forall(votes(node, ?party, ?votes), total := total + ?votes);
25    (?votes / total) * 100;
26
27  branch(node, action(next, dist)):
28      { assert(visited(node));
29        display("enter ", node, eoln);
30        forall({link(node, ?next, ?dist), not(prohibited(?node, ?next)}, action(?next, ?dist));
31        display("exit  ", node, eoln);
32        retract(visited(node)) };
33
34    branch_and_bound(start, stop):
35      { bound: 999999999;
36        traverse(node, sum):
37          if(isfree(node),
38             if(sum<bound,
39                if(node=stop,
40                   { bound:= sum;
41                     display("reach ", node, " after ", bound, eoln) },
42                   branch(node, traverse(next, sum+dist)))));
43        traverse(start, 0);
44        display("from ", start, " to ", stop, " is ", bound, eoln) }
```

Figure 1: Code for the Branch and Bound example

Evaluation of the example program would entail a number of switches between the P- and L-parts of the combined interpreter. Such a switch would occur when either part encounters an abstract expression it doesn't know how to handle. Take line 32 in the example where the P-part would be busy applying the traverse function to some arguments: here it encounters an application of the function `if` with a predicate that is actually a logic pattern. The P-part doesn't know how to evaluate such an expression but the L-part does. Other such switches need to happen at lines 23, 25, 27 etc. Also note that before either part returns the evaluation result of one such switch, a number of other switches may occur back and forth. This for example happens at line 25, where the logic pattern `forall` is evaluated: here a function application of `action` occurs. The code on the same line goes even further in that it also makes use of the `prohibited` predicate, which in turn at line 14 makes use of the `percentage_votes` function; the latter again involves logic code at lines 17 and 19.

Now why should such switching occur through coroutines rather than simple subroutine calls? Is it not possible for the P- and L-parts to simply call each other's "evaluate" routine? In the idealized case where both evaluate^P and evaluate^L are implemented in the same high-level language, use the same memory model, and are fully reentrant, simple subroutine calling would suffice. In practice the two interpreters are implemented in a portable low-level language such as C, use their own memory models with a specific garbage collection algorithm to represent programs and values, and use a particular execution model such as continuation based execution. Reentrancy of certain procedures, such as evaluate, is only guaranteed for certain other procedures of the same interpreter. For instance, for efficiency reasons the evaluate^P might have been implemented such that the current environment is stored in a static variable, instead of passing it around as an extra argument. When other procedures of the interpreter need to call evaluate without making changes to the current environment they will need to explicitly save it before and restore it after the call. It is highly unlikely that the environment save/restore wrapper for function application can be generalized to fit the evaluation of a nested expression^L. On the other hand, the evaluation of an expression^P nested in the latter will require the ececution environment^P to be available in its original state. The point is that in general these transfers of control are more than simple function calls - in the example they occur during a crucial phase of the interpretation process: parameter binding and unification respectively. If we want to preserve as much as possible from the original evaluate^P and evaluate^L we will need to separate the control structures for both parts of the LP-interpreter as much as possible. These requirements are easily satisfied using a coroutine approach: in our case the dispatchers for both original interpreters need to be extended to perform a coroutine transfer to the partner evaluator in the case where a non-own tag is encountered.

Although it can be argued that for very specific situations, the coroutine communication between two symbiotically linked interpreters will revert to a subroutine call, this not generally true. In order to maximize reuse of the code of the two participating interpreters, a coroutine framework seems to be a viable and attractive solution.

5 Conclusion

The discussion in the previous section was inspired by a need for reuse. However, we can go one step further and examine whether we cannot use a software engineering approach to language engineering: establish interpreters for specific programming paradigms as a kind of reusable components and wire them together using a coroutine based glue language. At the very least this should illustrate that software meta-engineering and meta-software engineering are but two sides of the same coin. Experiments are underway.

6 References

[ASPECT] Is Domain Knowledge an Aspect?
M. D'Hondt, T. D'Hondt: ECOOP Workshops 1999: 293-294
[PICO] http://pico.vub.ac.be
[SCHELOG] http://www.ccs.neu.edu/home/dorai/schelog/schelog.html
[SICP] Structure and Interpretation of Programming Languages
H. Abelson and G. Sussman: MIT Press (1996)