Domain-Specific Languages in Few Steps
The Neverlang Approach

Walter Cazzola

Department of Informatics and Communication
Università degli studi di Milano
cazzola@dico.unimi.it

Abstract. Often an *ad hoc* programming language integrating features from different programming languages and paradigms represents the best choice to express a concise and clean solution to a problem. But, developing a programming language is not an easy task and this often discourages from developing your problem-oriented or *domain-specific* language. To foster DSL development and to favor clean and concise problem-oriented solutions we developed Neverlang.

The Neverlang framework provides a mechanism to build custom programming languages up from *features* coming from different languages. The composability and flexibility provided by Neverlang permit to develop a new programming language by simply composing features from previously developed languages and reusing the corresponding support code (parsers, code generators, . . .).

In this work, we explore the Neverlang framework and try out its benefits in a case study that merges functional programming *à la* Python with coordination for distributed programming as in Linda.

Keywords: Development Tools, Language Design and Implementation, DSL, Composability, Modularity and Reusability.

1 Introduction

Nowadays, several and widely used programming languages support different programming paradigms, such as Erlang [25], Python [19] and Scala [21]. Such a design choice is an attempt to remedy to the lack of conciseness that is often manifest in a traditional general-purpose language. To have at disposal only a programming paradigm is too rigid and forces to write code that awkwardly solve specific problems, e.g., try to imagine how should be to write generic sorting algorithms without first-order functions, function objects or templates.

Even if multi-paradigm programming languages put at disposal several programming paradigms, the different programming models might not offer a concise way to express the desired solutions and often their complexity could be excessive with respect to the requirements. Moreover to let coexist different programming paradigms means to compromise some of the functionality of one or the other paradigm and such a compromise could endanger the expected benefits. Last but not least you are still framed in the language designer's design and what you need could be still missing.

Another approach to favor the conciseness of the written program is based on extending a programming language via an external library to fulfill the desired missing features (e.g., the ODBC library). In general, this approach is tailored on the desired feature and so cleaner but the adopted syntax and the integration of the introduced feature with the rest of the programming language are far from optimal and sometimes its usage could result cumbersome. Moreover, the API provide a more complex interface if compared with the same feature supported by the language [17] (e.g., compare iterating on a collection in Java with or without the *foreach* construct) and in statically typed languages its integration cannot be as seamless as desired (e.g., some casts could be necessary). To clear up these issues look at the advantages of LINQ (a DSL for DB connectivity) over ODBC based approaches [16].

Other minor issues that should foster the provision of *ad hoc* programming languages are related to efficiency and extensibility. A general-purpose programming language usually provides several programming features but some of them might be unnecessary or redundant (e.g., Python's map/filter/reduce and list comprehensions) and contribute to make the language over complicate to learn and use. Mainstream programming languages have a poor support for extensibility [3, 7] and those that are designed to be extensible (e.g., Lisp, Scala) did not gain a wide acceptance [2] or are really inefficient (e.g., parser combinators in Scala[1]).

The ideal solution to get conciseness, efficiency and extensibility would be to develop a *domain specific* programming language by combining only those features really necessary to solve the target problem, reusing the definition and implementation from other programming languages. This would allow different paradigms to be freely mixed in a fine grained manner and to speed up the design and implementation of a new *ad hoc* programming language. To this respect we have developed the Neverlang [5,6] framework that permits to design new languages in terms of features of other programming languages and to fast generate an interpreter/compiler for such language by reusing pieces of the compilers/interpreters implementing such features.

The rest of the paper has the following organization. Sect. 2 introduces the Neverlang framework whereas some details on the implementation are in Sect. 4. Sect. 3 explores the potential of Neverlang by showing a case study focused on the creation of a concurrent/functional programming language; in particular it shows how to mix two language definitions to create a new language and how to change the behavior of an existing language. Sect. 5 provides a critical analysis of some related works pointing out the differences and the innovations introduced by the Neverlang framework. Last but not least, in Sect. 6 we draw our conclusions.

2 The Neverlang Framework

The Neverlang [5, 6] framework is inspired by HyperJ's [23] multi-dimensional separation of concerns and basically reflects the fact that programming languages have a modular definition and each language feature can be easily added to or removed from

[1] `http://scala-programming-language.1934581.n4.nabble.com/Performance-of-Scala-s-parser-combinators-td3165648.html`

the language. Ideally, the design of a programming language should consist of selecting a set of features (building blocks) from existing languages and composing them together. The whole structure of the compiler/interpreter is the result of composing the code necessary to compile/interpret each single feature. The Neverlang framework realizes this vision and provides a language for writing the building blocks and a mechanism for composing the blocks together and generating the compiler/interpreter of the resulting language.

2.1 Neverlang at a Glance

In the next we describe the basic elements and concepts introduced by Neverlang and the composition model behind the approach.

Basic Framework Concepts. In our approach we exploit the vision that a programming language is defined in terms of its features (e.g., types, statements, relationships, and so on) and such features can be formally described in isolation (as productions of a grammar) and composed to form the language structure (syntax). Traditional compiling techniques [1] perform some transformation on such description that brings forth to the interpretable/compiled code. A complete compiler/interpreter built up with Neverlang is the result of a compositional process involving several basic units describing the language features and their support code.

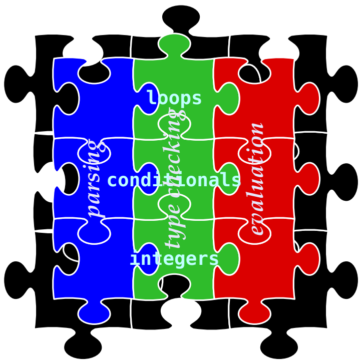


Fig. 1. Sectional DSL

These basic units are called *modules*. Each module encapsulates a specific feature, such as the syntactical aspect of a loop, and thus it is bound to a precise *role*, the syntax definition in our example. The role category determines where a module will be attached to. Roles can be interpreted as *dimensions* and are bound to the phase of the compilation and interpretation process. `syntax`, `type-checking` and `evaluation` are examples of key roles but they are not the only and fixed set of roles: the user can define new roles (in the language definition) associated with specific compilation phases and he can define the whole compilation process in terms of the defined and included compilation phases giving their relative order, i.e., specifying what phase precedes what (through the keyword **`roles`**).

Finally, modules regarding the same language structure but with different roles are grouped together in *slices*. The final language is simply the result of the slice composition. To some extent, we can say that slices are *orthogonal* to roles: the former are a collection of modules that compose the same feature, the latter are a collection of modules regarding the same compilation/interpretation phase. In this scenario designing a domain specific language consists of defining a set of slices and composing them together. In Fig. 1 is depicted the general multi-dimensional structure of a language developed by using Neverlang, the colored jigsaw pieces are modules, those in the same row contribute to describe the same feature and are part of the same slice whereas their color specifies which role they play. Neither the number of modules composing each slice must be always the same nor a module for each role must be present in a slice.

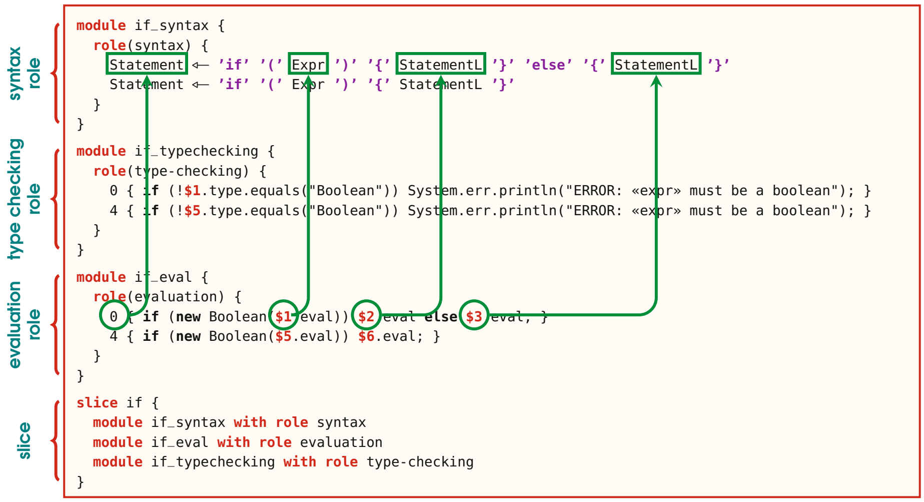
To plug a slice in, we need a mechanism to precisely select the insertion point inside the compiler/interpreter. The process for selecting these insertion points, or join points in aspect-oriented parlance, is grammar driven: they correspond to the nonterminal symbols of the grammar; a grammar that dynamically grows as new slices are plugged in. In our case the code to be introduced at the join points, advice in aspect-oriented jargon, participates to define/implement the compiler/interpreter of the new language and consists of the grammar productions (in the syntactic module) with the related semantic action routines (in the other modules). The Neverlang approach to compiler/interpreter building is *symmetric* [13]. The DSL is a sort of patchwork of only those features selected to be part of the language and its implementation is not achieved by modifying an existing compiler/interpreter but built up from the implementation of the single feature provided by the corresponding slices; that is, the compiler/interpreter is the result of the slice composition. The composition specification defines the grammar join points and its advice. A complete compiler/interpreter reifies its grammar join points, so that it can be subsequently extended with new productions. A pleasant effect of symmetric composition is that many slices can be easily reusable by different DSLs.

To support the various compilation/interpretation phases, the developer may need some ancillary structures or services that concerns the whole compilation process affecting all the other modules crosswise. Simple examples are the *symbol table* and the code to deal with the memory management. A slightly different form of slice called *endemic* supports this kind of behavior. The fields and methods defined in an endemic slice are accessible by all modules in the Neverlang program independently of the compiling/interpreting phase. Adding/replacing an endemic slice permits to easily redefine the whole behavior of the compiler/interpreter (more on this in Sect. 3.4).

Modules and Slices Definition. Listing 1 shows the Neverlang implementation for the *if-else* conditional construct. Three roles (`syntax`, `type-checking` and `evaluation`) are involved; each role is defined in a separate module (that could be written in separate files as well) and combined together in the `if` slice.

Syntactically, the *if-else* construct can be defined by two productions: the first defines the complete case with both branches and the second defines the case without the `else` branch (see [1]). Such productions are defined in a module with the role `syntax`. Each production is composed by *terminal*s (surrounded by `'`) and *nonterminal*s (e.g., `StatementL`). These productions are bound to the nonterminal `Statement`, other productions bound to such nonterminal can be defined in the `syntax` module of other slices. Obviously nonterminals used in the right side of the productions and unbound in the module (such as `StatementL` and `Expr`) must become bound in the slice composition phase where other slices with productions bound to them come into play; otherwise the grammar will be incomplete and the compiler/interpreter could not be generated.

All the other kind of modules (in our example those with role `type-checking` and `evaluation`) add *semantic action*s to the grammar rules defined in the module with `syntax` role (see *syntax-directed translation* in [1]). Each of these modules associates to the nonterminals the semantic actions necessary to carry out the corresponding compilation/interpretation phase — e.g., the semantic actions in the module with role



```
module if_syntax {
  role(syntax) {
    Statement ← 'if' '(' Expr ')' '{' StatementL '}' 'else' '{' StatementL '}'
    Statement ← 'if' '(' Expr ')' '{' StatementL '}'
  }
}
module if_typechecking {
  role(type-checking) {
    0 { if (!$1.type.equals("Boolean")) System.err.println("ERROR: «expr» must be a boolean"); }
    4 { if (!$5.type.equals("Boolean")) System.err.println("ERROR: «expr» must be a boolean"); }
  }
}
module if_eval {
  role(evaluation) {
    0 { if (new Boolean($1.eval)) $2.eval else $3.eval; }
    4 { if (new Boolean($5.eval)) $6.eval; }
  }
}
slice if {
  module if_syntax with role syntax
  module if_eval with role evaluation
  module if_typechecking with role type-checking
}
```

Listing 1. Simple `if-else` module and slice

`type-checking` are used during the type checking phase. The nonterminals are identified through their position in the productions numbering with 0 the top leftmost nonterminal and incrementing by one left-to-right and up-to-down all nonterminals independently from any repetition and for the whole set of productions defined in the slice. In our example, the module with role `evaluation` defines two semantic actions. The first enriches the head of the first production (position 0) and simply tests the evaluation of the boolean expression associated to the nonterminal in position 1 (`Expr`) and, accordingly to that, respectively evaluates the nonterminal in position 2 or 3 through their `eval` attribute (such associations are made explicit by arrows in the listing). The second action behaves similarly but refers to the case without the `else` branch and it is associated to the head of the second production (position 4). Semantic actions are anchored to a nonterminal; if it is the head of the production, the action evaluation is carried out after the evaluation of the semantic actions in the right part of the production and in its derivation (*postfix* evaluation); otherwise it is done before (*prefix* evaluation).

The semantic actions are basically pieces of Java code that access attributes computed during the current (or previous) compilation/interpretation phases. What the attributes are and how they are transmitted from a module to another derives directly from how the *syntax-directed translation* mechanism [1] works. Attributes are accessed through the nonterminal (by its position prefixed by `$`) they refer to, e.g., `$1.eval` is the `eval` attribute of the `Expr` nonterminal in the first production of our example. To make clear how the slice composition takes place, we can make a parallel with aspect-oriented parlance and consider a nonterminal as a sort of join point where productions and semantic actions are woven during the generation of the compiler/interpreter for the language. Finally, the keyword **`slice`** permits to select the modules that will compose our slice and to specify which role such modules play in the slice. Note that a role can be played by only one module in each slice.

```
# the master process defines the tasks for the slaves and collects the results.
process main { # distribute the jobs to the slaves and collect the results
  rLenA = args[0]              # A's and B's dimensions are passed as args
  cLenA = args[1] rLenB = args[2] cLenB = args[3]

  # A and B are randomly generated.
  A = [[random(10) for y in range(cLenA)] for x in range(rLenA)]
  B = [[random(10) for y in range(cLenB)] for x in range(rLenB)]

  # calculates B^T in a list, then converts it back to a matrix
  B = [B[y][x] for x in range(cLenB) for y in range(rLenB)]
  B = [[B[x] for x in range(y*rLenB, rLenB+y*rLenB)] for y in range(cLenB)]

  # drops the tuples <A_i, B_i^T>, x and y help in retrieving the results.
  [out("data", A[x], B[y], x, y)
     for x in range(rLenA)
       for y in range(cLenB)
  ]
  out("tuples", cLenA*rLenB)
  cnt=rLenA*cLenB

  # create an empty matrix for the result
  C = [[0*y for y in range(cLenB)] for x in range(rLenA)]

  tot=0                                         # collecting the results
  [C[x][y] = tot for i in range(cnt) if in("result", ?x, ?y, ?tot)]
  print(C)
}

# client (slave) calculates and drops in the tuple space the multiplications
process client {
  tot = 0  len = 0
  do {                        # search for non calculated couple of rows
    in("tuples", ?len)
    if (len != 0) {
      out("tuples", len-1)
      in("data", ?A, ?B, ?x, ?y)
      [tot = tot + z for z  in
          [A[i]*B[j] for i in range(len(A)) for j in range(len(B)) ]
      out("result", x, y, tot)
    }
  } while(len == 0)
}
```

Listing 2. Cooperative matrix calculation in the Linda+Python language

3 Neverlang at Work

In the next we present a case study that shows how Neverlang eases to mix up features from different programming languages to form a new one.

3.1 Case Study: Linda+Python

Nowadays multi-core computers are on the rise and the opportunity to program them as a parallel computer and possibly to communicate through their shared memory is coming into the limelight. Some decades ago, Gelernter *et al.* [4,11] introduced Linda, a parallel programming model based on shared memory, called the *tuple space*, to support inter-process communication. The tuple space approach could become topical again in the context of multi-core programming.

Linda is a coordination language with a very limited set of concepts (only six primitives: **in**, **inp**, **rd**, **rdp**, **out** and **eval**) that needs to be embedded in a Turing complete programming language to be useful and usable. Normally, Linda is integrated into another programming language either by modifying the existing compiler for the host language or by using an external library (as in JavaSpaces [10]). As we explained in the introduction both approaches have drawbacks: modifying a compiler is a time-consuming and error-prone task whereas an external API could badly fit in the original programming language and its use could be complicated and cumbersome. Neverlang can be used to overcome these issues by simply supporting the compiler/interpreter generation for a domain specific programming language. In particular, the coordination language Linda is merged to the functional characteristics of Python (in particular Python's *list comprehension*s) [18] to form the so-called Linda+Python programming language. This is realized by writing down the necessary slices that permit to support (portion of) the two languages by merging them together to define the new language. Of course, we maximize the benefits when the languages to mix up are already implemented in Neverlang and their slices can be reused.

The result of the process is a framework (compiler and interpreter) that permits to compile and to interpret programs written in the Linda+Python idiom. As an example, listing 2 implements a distributed and shared memory-based version of the matrix multiplication algorithm in Linda+Python: keywords from the Linda language are blue-colored whereas the red-colored are Python keywords. The program follows the master/slaves paradigm; two kind of processes are involved: `main` and `client`. The former (master) stores row and column couples in the tuple space and waits for the result. The latter (slave) looks in the tuple space for such kind of couples, multiplies each couple element by element and puts back in the tuple space a tuple with the sum of such products. Note that more than one `client` process can be launched without any conflict to speed up the process.

3.2 Linda+Python Building Blocks

In the long term, i.e., when the Neverlang framework will support enough programming languages, developing the support for the Linda+Python programming language will simply consist of reusing the slices from Python and those from Linda regarding the features of interest and writing some glue code to merge up the whole system. Since we are still far from such a situation we have developed the necessary slices as well.

Linda. The Linda implementation in Neverlang should, at least, provide a slice for each Linda primitive, a slice to support *tuple*s and *anti-tuple*s and the support for the *tuple space*. The slices to implement the primitives and the *tuple*s and *anti-tuple*s do not introduce particularly relevant concepts. The implementation of the *tuple space* is more interesting since it does not introduce any piece of new syntax but it is just a sort of data structure used by the other primitives; it is the perfect example of feature endemic to the rest of the programming language and it is implemented in the endemic slice `TupleSpace` (Listing 3). In this case, the `type-checking` phase does not only support the classic type checking but also the tuple matching to extract tuples from the *tuple space*. As for the *tuple space* management, the `TupleSpace` relies on an external Java package (cf. the `TupleSpaceThread` class in Listing 4). The endemic slice is used by the other slices through the syntax.

```
slice TupleSpace {
  decl {
    import TupleSpace.TupleEntry; import TupleSpace.TupleSpaceThread;
    TupleSpaceThread ts = TupleSpaceThread.getInstance();                    // tuple space

    Hashtable<String,String> getMatches(ArrayList<String> val,ArrayList<String> types,boolean del) {
      TupleEntry res, te = new TupleEntry(types, val);                       // anti-tuple
      if(!del) res = ts.getMatches(te);                                      // read operation
      else res = ts.getMatchesAndRemove(te);                                 // in operation
                                ...
    }

    void addEntry(ArrayList<String> values, ArrayList<String> types) {
      ts.addEntry(new TupleEntry(types, values););
    }
  }
  module TupleSpace with role endemic
}
```

Listing 3. The endemic slice to support the tuple space

```
public class TupleSpaceThread implements TupleSpaceInterface {
  private static TupleSpace ts;
  public static TupleSpace getInstance() {
    if (ts == null) ts = new TupleSpace();
    return ts;
  }
  public void addEntry(TupleEntry te) { ... }
  public TupleEntry getMatch(TupleEntry te) { ... }
  public TupleEntry getMatchesAndRemove(TupleEntry te) { ... }
}
```

Listing 4. The external `TupleSpaceThread` library

```
«slice name».«operation name»(«arguments»).
```

The endemic slices are always available and do not need to be imported.

Python. In the case of Python, we are just interested in its list datatype and in the comprehension mechanism. Python lists are heterogeneous and dynamically typed. Dynamic typing forces to postpone the list type evaluation to the evaluation phase and in the case of Linda+Python example to have a mixed type checking: static for the Linda part and dynamic otherwise. The Neverlang framework permits to suspend and to resume a phase on part of the AST; this feature enables the system to postpone the type checking at the evaluation phase.

Due to space limitations, we cannot report the whole slice dealing with the list comprehension feature but we are interested in showing how the suspend/resume mechanism works. Listing 5 shows how to support dynamic checking in a `for-each`-like construct that iterates on a heterogeneous list and executes an expression on every element. During the `type-checking` phase, the expression associated to the `SimpleExpression` nonterminal cannot be type checked because it is (presumably) bound to the identifier (described by the `Identifier` nonterminal) whose type is not available before the evaluation and potentially it can change at each loop since Python's lists are not bound to a single type. The `type-checking` phase for the `SimpleExpression` nonterminal must be suspended by using the special function **`$suspend`**. During the `evaluation` phase, the type and value of the elements of the list are stored in a *variable table* that can be used

```
module ForEach {
  role(syntax) {
    ForEach <- SimpleExpression 'for' Identifier 'in' List
  }
  role(type-checking) {
    1 { $suspend; }
         ...
  }
  role(evaluation) {
    0 {
      String listValue = $3.eval;
      String[] values = VarTable.getValues(listValue);
      String[] types = VarTable.getTypes(listValue);
      for(int i=0; i<values.length; i++) {
         VarTable.putValue($2.eval, values[i]);
         VarTable.putType($2.eval, types[i]);
         $1.resume("type-checking");
         $1.eval;
      }
    }
  }
}
```

Listing 5. Postponed type checking in the implementation of the comprehensions

```
module Inc {
 role(syntax) { SimpleExpression <- Identifier '++' }
 role(type-checking) {
   0 {            // type checking resumes here
     if !(VarTable.getType($1.eval).equals("int")) Logger.printError("Invalid type!");
     else $0.type = "int"
   }
 }
 role(evaluation) {
   0 { VarTable.putValue($1.eval, VarTable.getValue($1.eval)+1); }
 }
}
```

Listing 6. Module `Inc` to implement the increment operation

by the type checker when resumed (call to the **`$resume`** special function). Given that the `SimpleExpression` nonterminal expands into the increment operation (defined by the `Inc` module, Listing 6), the type checking process resumes in the semantic action associated to the head in such module, i.e., the place where the phase is resumed depends on the program we are compiling. In the `type-checking` and `evaluation` phases the values stored in the variable table before the **`$resume`** is used; the variable table is implemented by the `VarTable` endemic slice as a hash table.

3.3 Building Linda+Python Up

Once the necessary slices have been developed (or selected if already existing), we have to define (through the **`language`** statement) how such slices are composed together to form the new Linda+Python programming language; such definition will drive the Neverlang framework in the building of a compiler/interpreter for the new language.

Being the compilation phases syntax-driven, most of the issues the developer has still to face are the syntactical conflicts that could rise by composing modules with role

```
language Linda+Python {
  slices Main BoolOp Sum Process Mul Out Read InP ReadP Eval Length Id ListsOp WildCardExpr Boolean
     Int Assign ListsAccess Range In CompVarTable Tuple Args Expr Rand Var VarTable Comprehension
     DoWhile If OpTable MulTable ListAssign TupleSpace IntOp Lists Print BooleanOpTable SumTable
     ListsTable CompFor
  roles syntax < type-checking < evaluation
}
```

Listing 7. Linda+Python Definition

`syntax` defined by different development teams. A quite common example of this problem occurs when you compose statements such as `if` and `do-while`, whose productions have different left-hand nonterminals rather than a common one as `Statement`; such problem requires some glue code to redefine or to make uniform the productions. Neverlang avoids further conflicts in the other phases since the semantic actions are associated to the nonterminal position and this is related to a specific production independently of the slice composition.

Listing 7 shows the language composition in our case study; such a piece of code just lists which slices should be composed and in which order the compiling phases occur (expressed through the keyword **`roles`**).

3.4 Flushing Flexibility Out

The tuple space implementation and the distribution provided in Listing 3 are quite naïve; Linda's processes are implemented as threads and the tuple space resides in the data area common to all threads. Of course, this is just a proof of concept but it permits to show another peculiarity of the Neverlang approach: how easy it is to evolve a programming language (more on DSL maintenance in Neverlang can be read in [5]). Switching from the current thread-based implementation to a more distributed RMI-based one is just a matter of substituting the `TupleSpace` slice with another slice supporting the desired implementation while retaining the interface.

Listing 8 shows the endemic slice `TupleSpaceRMI` that will replace the thread-based implementation of the tuple space. The instance of `TupleSpace` does not reside in the common data area anymore but it is accessed through RMI as a remote object. Other minor changes are not showed due to sake of space but the access methods are *synchronized* and the `Tuple` are *serialized* to be stored into or retrieved from the tuple space.

This kind of change is particularly easy since it just affects the run-time environment of a running program and not its syntax and semantics and (with some care for consistency) it could be done at run-time as well. Of course any kind of language evolution can be easily taken in consideration but often this affects the source code as well due to changes to the language syntax. Note that similar flexibility can be achieved also with library based solutions but with a less clean syntax; moreover it is hard to change or extend features whose implementation is less self-contained and library based. In [5] are shown more elaborated kind of evolutions.

The complete case study can be downloaded from the Neverlang web page:

`http://cazzola.dico.unimi.it/neverlang.html`

```
slice TupleSpaceRMI {                  // TupleSpaceRMI differs from TupleSpace in
            ...                        // the way the tuple space is retrieved
  decl {
    TupleSpace ts = connect();

    TupleSpace connect() {
      try {
        return (TupleSpaceInterface)LocateRegistry.getRegistry(Args.hostname).lookup("TupleSpace");
      } catch (Exception e) { ... }
            ...
    }

    public class TupleSpaceRMI implements TupleSpaceInterface {
                   ...
      public static void main(String args[]) {
        try {
          TupleSpace ts = TupleSpace.getInstance();
          TupleSpaceInterface stub = (TupleSpaceInterface)UnicastRemoteObject.exportObject(ts, 0);
          LocateRegistry.getRegistry().rebind("TupleSpace", stub);
        } catch (Exception e) { e.printStackTrace(); }
      }
    }
  }

  module TupleSpaceRMI with role endemic
}
```

Listing 8. RMI-based tuple space implementation

4 Neverlang Close-up

Basically, the idea behind the Neverlang framework is to compose the slices listed in the **`slices`** section of the **`language`** statement and to exploit the syntax-directed translation [1] approach on the context-free grammar that results from the `syntax` module composition which is then decorated with the semantic actions specified in the remaining modules.

The resulting compiler/interpreter is mainly composed of two parts: i) a front-end that parses the source files written in the new language and generates the classes that will compose the *abstract syntax tree* (AST) and the AST itself and ii) a type-driven back-end that attaches (through aspect-oriented programming) the semantic actions specified by the developer to the classes composing the AST and traverses the AST to carry out all the compilation/interpretation phases.

Front-End Generation. To render the language definition extensible and sectional we adopted the *parsing expression grammar*s (PEGs) [9] and the Rats! [12] an extensible parser generator that works on these kind of grammars. PEGs look similar to context free grammars but they are not ambiguous: if a string parses, it has exactly one valid parse tree and it is always possible to write a recursive-descent parser running in linear time (the pakrat parser [12]).

The compiler generation procedure starts by collecting all grammar productions contained in the modules with role `syntax` and by translating them in Rats! modules that define the parser for the new language. The tool builds a class (a sort of empty skeleton to be filled later by the back-end) for each nonterminal of the generated grammar; such classes will be used to instantiate the nodes of the AST accordingly to the grammar of the defined language.

The most natural way of representing an AST is to model the language constructs as a class hierarchy with general abstract classes like `Statement` and `Expression`, and specialized concrete classes like `Assignment` and `AddExpression`. In our case, all nonterminals are modeled as abstract classes and the productions permit to specialize them in concrete subclasses. All nonterminals have a common superclass, called `AbsNode`, containing the fields and methods common to all nonterminals; this is refined in concrete classes representing the nonterminals and containing their own inherited and synthesized attributes [1].

Semantic Back-End Generation. The semantic actions associated to a nonterminal are appropriately injected into the AST node representing such nonterminal. Once all the semantic actions related to every compilation/interpretation phase are injected, the compilation/interpretation process can start. Each phase is associated to a specific `visit()` method and will be carried out during the AST traversal by calling such method on each node.

The evaluation carried out at each node is both type-driven (i.e., associated to the corresponding nonterminal) and context-driven (i.e., related to the semantic action associated to the position of the nonterminal in a given production; a nonterminal can occur in several positions and in each position can be decorated by a different semantic action). Therefore, the code of the method invoked during the visit will change on a per node basis (Polyglot [20] has similar necessities solved by delegation). AspectJ [15] represents the perfect tool to realize the required context-driven adaptation of the AST nodes. The code of the semantic actions associated to each nonterminal is automatically woven into the method invoked during the tree traversal accordingly to the type of the node, to the node position in the AST (and consequently the position the corresponding nonterminal has in the applied production) and to the compilation/interpretation phase (that is, the role we are effectively playing). By using this approach, semantic roles are implemented without modifying the classes of the AST nodes and the whole role can be easily plugged and unplugged. This context-adapting `visit()` method implements a sort of aspect-oriented *modular visitor pattern* [22] that permits to avoid the well-known *expression problem* [27]; with respect to Oliveira's [22] proposal, this has also the benefit of reducing the necessary casts.

In detail, a pool of aspects is created for each role (or compilation/interpretation phase if you prefer). Such aspects wrap up the pieces of code that should be attached to the method called during the AST traversal. Each collection of aspects contains also a special element called *driver aspect* that drives the entire compilation/interpretation phase and the switch from a phase to the following; the mechanism is quite simple: the generated compiler/interpreter has a hook in its main program (a dummy method invoked just after the tree construction, that permits to put in evidence a join point). This hook is used by the driver aspect as an anchor where to hook up the AST `visit()` method for a given compilation/interpretation phase (woven before the method call) and the code to switch to the next phase (woven after the method call). At each phase, the AST traversal depends on a flag attached to each node: if the flag is unmarked the node and its children are skipped during the visit; normally all the nodes are marked for the visit. The **`$suspend`** primitive unchecks the corresponding node for the current phase and a special aspect is created to be used when resuming; the **`$resume`** primitive represents a join point where such aspect is woven to permit the belated visit of the AST. Methods and fields defined in endemic slices are wrapped in static classes and imported by all the generated aspects and classes. These fields are initialized before the AST traversal starts and the implemented services are available during the whole compilation procedure.

To establish an order among the compilation/interpretation phases as expressed by the **roles** keyword, we exploit the advice precedence feature of AspectJ that permits to specify in which order to apply the advices matching the same join point. In particular, in the main procedure we set the order in which the *driver aspect*s are woven into the dummy methods.

5 Related Work

Several works share Neverlang's goals. JastAdd, xText and Polyglot are the most pertaining.

JastAdd. The JastAdd [8, 14] system enables open modular specifications of extensible compiler tools and languages. JastAdd is an extension to Java that supports a specification formalism called rewritable circular reference attributed grammars.

JastAdd and Neverlang share a very similar object-oriented implementation of the AST [8]. Moreover, they both adopt aspect-oriented programming to extend the language behavior by injecting methods and fields in the AST nodes. On the other side, in Neverlang the AST nodes and their connections come after the grammar productions whereas in JastAdd they can be user-defined granting a major flexibility but the generated code can bloat.

JastAdd adopts *reference attributed grammars*, i.e., a semantic action in q can refer to an attribute of an unrelated nonterminal r. PEGs, adopted by Neverlang, do no support this feature but it can be simulated by saving r in an external data structure (through an endemic slice) during the AST visit (as we do to deal with the attributes).

In JastAdd each declared behavior rewrites the AST tree nodes giving the opportunity to add or delay a phase of compilation; behaviors are similar to Neverlang roles. Even if Neverlang's modularity (roles) is not limited to compiler phases but straddles the whole compilation/interpretation process via the endemic slices.

Polyglot. Polyglot [20] is an extensible compiler framework that supports the creation of compilers for Java-like languages. Polyglot relies on an *extensible parser generator* that permits to express the language syntactical extensions as changes to the Java grammar.

Polyglot extensibility is supported by *delegation*. Each compilation phase is supported by a delegate object present in each AST node type; the delegate object is appropriately replaced in each extension.

Neverlang and Polyglot share similar goals, i.e., supporting the development of syntactical and semantical extensions to a programming language but Polyglot is limited to Java. Besides, Polyglot extensions are just source-to-source translations from the extended language to pure Java. Modularity and reusability are issues that Polyglot does not face.

xText. xText is an Eclipse plug-in that provides a framework for the development of domain-specific languages. It is tightly integrated with the Eclipse modeling framework [26] to provide a language-specific IDE.

Like JastAdd the user is free to define the relation between grammar productions and AST nodes but each parser rule will create a new node in the AST. The language meta-model describes the structure of its AST.
xText generator leverages the *modeling workflow engine* from Eclipse modeling framework technology and the code is generated from the meta-model created by the parser; the meta-model is similar to the Neverlang semantic back-end.

The framework gives the opportunity to reuse existing grammars and existing meta-models to implement the back-end for different languages. However the framework seems oriented to infer a model from a text and to translate it to an other model (*model-driven development*) rather than to create real compilers. A similar approach (model-driven) is also provided by Frag [28].

To recap, the main difference, that evinces from this comparison, is that Neverlang focuses on modularity and reusability of the compiler units; in Neverlang the developer can easily extend and mix existing languages to define new languages with working compilers/interpreters. Moreover, by compiling Neverlang programs we get real compilers/interpreters for programming languages completely independent of the language used to implement the compiler (Java in our case) and not (source-to-source, model-to-model, ...) translators that limit the implemented programming language to syntactic extensions of the host language.

6 Conclusions

In this paper we have introduced Neverlang: a framework to describe new programming languages as the composition of programming features from existing programming languages and to generate the compiler/interpreter for the new language by reusing previous implementations. Moreover we have shown how Neverlang can be used to mix up programming features from Python (*list comprehensions*) and Linda (*coordination*) to form a multi-paradigm programming language and showed how its implementation can be easily changed from a thread-based tuple space to a RMI-based one.

Currently an incremental parser (similar to the PetitParser [24]) is under development, the idea is to drop the PEG-based parser in favor of a more flexible parser that will permit to avoid of regenerating the whole parser when a slice is added or removed from the language. This would permit to change the behavior of a running system, for example, in the Linda+Python case study we could change the tuple space implementation from one version to another on-the-fly. Also the implementation of some well known languages like Java and Python is under development as well as the improvement of the composition mechanism to support a finer decomposition for the feature implementation and to ease the mix up of different interpretation/compilation philosophies (e.g., monadic and traditional interpretation).

Acknowledgments. The author wishes to thanks Ivan Speziale and Davide Poletti that worked on the Neverlang implementation; without their help Neverlang would be a nice *empty* box.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading (1986)
2. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Proceedings of the 5th International Conference on Software Reuse, Victoria, BC, Canada, pp. 143–153. IEEE Computer Society (June 1998)
3. Bravenboer, M., Visser, E.: Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, BC, Canada, pp. 365–383. ACM (October 2004)
4. Carriero, N., Gelernter, D.: Linda in Context. Commun. ACM 32(4), 444–458 (1989)
5. Cazzola, W., Poletti, D.: DSL Evolution through Composition. In: Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE 2010, Maribor, Slovenia. ACM (June 2010)
6. Cazzola, W., Speziale, I.: Sectional Domain Specific Languages. In: Proceedings of the 4th Domain Specific Aspect-Oriented Languages, DSAL 2009, Charlottesville, Virginia, USA, pp. 11–14. ACM (March 2009)
7. Ekman, T.: Extensible Compiler Construction. Phd thesis, Department of Computer Science, Lund University, Lund, Sweden (2006)
8. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2007, Montréal, Québec, Canada, pp. 1–18. ACM (October 2007)
9. Ford, B.: Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, pp. 111–122. ACM (January 2004)
10. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley (1999)
11. Gelernter, D.: Generative Communication in Linda. ACM Trans. Prog. Lang. Syst. 7(1), 80–112 (1985)
12. Grimm, R.: Better Extensibility through Modular Syntax. In: Schwartzbach, M.I., Ball, T. (eds.) Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, PLDI 2006, Ottawa, Ontario, Canada (June 2006)
13. Harrison, W., Ossher, H., Tarr, P.: Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical Report RC22685 (W0212-147), IBM (December 2002)
14. Hedin, G., Magnusson, E.: JastAdd — An Aspect-Oriented Compiler Construction System. Science of Computer Programming 47(1), 37–58 (2003)
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
16. Marguerie, F., Eichert, S., Wooley, J.: Introducing LINQ. In: LINQ in Action. Manning (January 2008)
17. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain Specific Languages. ACM Comput. Surv. 37(4), 316–344 (2005)
18. Mertz, D.: Functional Programming in Python. In: Charming Python, ch. 13 (January 2001), http://gnosis.cx/publish/programming/charming_python_13.txt
19. Nielson, S.J., Knutson, C.D.: OO++: Exploring the Multiparadigm Shift. In: Prooceedings of ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages, MPOOL 2004, Oslo, Norway (June 2004)

20. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
21. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Aritma Press (2008)
22. Oliveira, B.C.d.S.: Modular Visitor Components: A Practical Solution to the Expression Families Problem. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 269–293. Springer, Heidelberg (2009)
23. Ossher, H., Tarr, P.:Hyper/J: Multi-Dimensional Separation of Concerns for Java. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, Ontario, Canada, pp. 729–730. IEEE Computer Society (2001)
24. Renggli, L., Ducasse, S., Gîrba, T., Nierstrasz, O.: Practical Dynamic Grammars for Dynamic Languages. In: Proceedings of the 4th Workshop on Dynamic Languages and Applications, DYLA 2010, Málaga, Spain (June 2010)
25. Sahlin, D.: The Concurrent Functional Programming Language Erlang – An Overview. In: Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany (September 1996)
26. Steinberg, D., Budinsky, D., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley (December 2008)
27. Torgersen, M.: The Expression Problem Revisited. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
28. Zdun, U.: A DSL Toolkit for Deferring Architectural Decisions in DSL-Based Software Design. Information and Software Technology 52(7), 733–748 (2010)