DEXTER and Neverlang: A Union Towards Dynamicity

Walter Cazzola
Department of Computer Science
Università degli studi di Milano
cazzola@dico.unimi.it

Edoardo Vacchi
Department of Computer Science
Università degli studi di Milano
vacchi@dico.unimi.it

ABSTRACT

In recent years, functional programming languages have been cross-pollinating the object-oriented world. C# is now including constructs and concepts that are typical of functional languages, such as first-order functions and lazy evaluation, and there is a rising interest in multi-paradigm languages, like Python and Scala. The tendency to contamination between different programming styles can be read as the symptom of a need for more flexibility and conciseness in general-purpose languages. This is especially true when the constructs that a programming language provides are felt like insufficient or inadequate to express the solution of a domain-specific problem. In fact, we expect a domain-specific problem to be better modeled using a language that is specific to that domain. The Neverlang framework is designed to assist developers in the implementation of their own problem-oriented languages, using a sectional approach that favors code reuse. Neverlang acts as a toolchain that puts in sequence the typical stages of language development (defining the grammar, the semantics, and compiling the result), and generates code to implement an interpreter or a compiler. Unfortunately, generated code cannot be updated progressively: it has to be regenerated each time the sources change. We developed an LALR(1) parser generator that can update the implementation of an existing parser on the fly, avoiding any code generation intermediate step. We called it DEXTER: the Dynamically EXTEnsible Recognizer. In this paper we present the integration of Neverlang with DEXTER.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Compiler Generators,Parsing*

1. INTRODUCTION

Linguistic theorists call *linguistic relativity* or *Sapir-Whorf hypothesis* the principle according to which the structure of a natural language would affect the way its speakers are able to conceptualize their world. Even though the hypothesis is still object of research in the field of linguistics, we know that the features of a programming language *do* influence the way developers are able to conceptualize a domain of interest [8, 13]. Programming languages should be designed to let developers express their intents in the most natural way: when they fail to meet this fundamental requirement, they are in fact actively working *against* the developer, who is then forced to work around the limitations of a tool that should help him in the first place. This is especially true when we think of different programming paradigms: for a paradoxical example, just think of how hard is doing object orientation in a purely imperative programming language.

Addressing this problem is not easy. There is a rising interest in programming languages such as Python [20] and Scala [17], that mix concepts from the object-oriented world with idioms that are typical of functional languages. Even languages that were born with pure object-orientation in mind nowadays tend to include functional constructs (e.g., C# delegates and LINQ [16] or Java's Project Lambda[1]).

Domain-specific languages (DSLs) deal with the problem from another standpoint: if we take for granted that no general-purpose programming language will ever be able to express and model every concept of a domain of interest in full and without compromises, then the solution is to actually design a language with that domain of interest in mind. If we resort again to an analogy with linguistics, where the same concept of DSL has been borrowed from, you can describe a chemical reaction in plain English, but it would require more words than using the appropriate lexicon. It is much easier to describe a domain-specific problem, if the language we employ is specific to that domain. In computer science, we call DSL a language that is targeted towards a *specific problem area*. For instance LaTeX is a typesetting DSL, `SQL` is a DSL for querying relational databases, and a `Makefile` uses a DSL to describe batch builds.

The Neverlang [5] framework assists developers in the design and implementation of a DSL by composing the implementation of features from different programming languages into the interpreter or the compiler of a new language. Neverlang is built around the concept of *slice*, that is, the smallest building block of which a language is made by. A single slice describes only one part of the syntax and the related semantics. For instance, in the Java language, one slice could define the `while` looping construct, another slice could define the `if` conditional construct, and so on. If we see a language as the composition of several slices, then, with Neverlang,

[1] JSR 335, `http://www.jcp.org/en/jsr/summary?id=335`

ICOOOLPS'12, June 11, 2012, Beijing, China

```
task {
  remove application.debug.old
  rename application.debug application.debug.old
}
task {
  backup access.error
  backup system.error
}
```

Listing 1: LogLang example source file.

one can choose slices from different programming languages and compose them in different ways to obtain a new DSL.

At the moment, Neverlang produces interpreters or compilers using code generation techniques, which are commonplace when dealing with parser generators and compiler construction. In the last year, we have been working on a different approach to parser generation that will enable Neverlang to define and alter languages *progressively* and even at runtime. In particular, the grammar portions of a slice could be plugged and unplugged to the parser on the fly. Full recompilations, which are the norm when using generative tools, would become unnecessary. Neverlang's core could then act as a library: it could be imported within any application and provide primitives for language manipulation. For instance, IDEs for language development could keep a test implementation in-memory and update it as the users write slices. Of course, modern language development environments (e.g., [18, 4, 9]), already provide with live testing capabilities, but, being usually backed by a traditional parser generator, this often involves reloading big chunks of code or even restarting the entire platform.

The prototype parser generator we developed is called DEXTER: the Dynamically EXTEnsible Recognizer. DEXTER is a library that generates LALR(1) parsers that can be modified incrementally and in-memory. In this paper we describe the Neverlang framework and, most importantly, the steps that are necessary to integrate DEXTER in Neverlang. The rest of this paper is structured as follows: in Section 2 we describe how Neverlang works, by the help of an example; we first analyze a simple input file, we then proceed to describe how Neverlang is implemented, and what is DEXTER's role in the framework. In Sect. 3 we detail the implementation of DEXTER, how it is integrated with Neverlang, and how it performs; in the last part, we discuss how we could enhance Neverlang further, by making the implementation fully dynamic. In Sect. 4 we draw our conclusions and briefly describe our projects for the future of Neverlang.

2. NEVERLANG IN ACTION

The Neverlang framework generates parsers or interpreters using a compositional approach. The basic units involved in the compositional process are called *modules*. A module usually encapsulates a single feature and serves one or more roles. *Role*s bind code in the module to a specific phase of the compilation process: the `syntax` role defines the grammar of a syntactic feature, the `endemic` role declares methods, objects or values whose scope should span across every other module. Users can define new roles and specify their execution order, with respect to the compilation phase. Typically, there will be at least an `evaluation` role, to perform semantic actions, bound to some grammar rule. Distinct, but related modules, with different roles, can be grouped together in *slices*.

A simple DSL for log rotation. In [6] Neverlang is introduced by the help of a simple DSL inspired by the Unix `logrotate` utility, that we called *LogLang*. The `logrotate` utility helps administrators in managing system logs by automatically removing, renaming, and making backup copies of these files. The tasks are specified in a configuration file, and the utility is scheduled to run periodically using `cron`. Instead of developing a program that reads a configuration file and executes the corresponding tasks, we promote a language-driven approach. A file written using LogLang is not a configuration file, but rather a small program for the LogLang interpreter. An example source file is given in Listing 1. The example shows two simple tasks. The first task rotates a debug log for an application named `application`: it deletes the old log and renames the current. The second task creates a backup copy for the specified logs. The modular approach allows any system administrator to define new statements and update the interpreter. Evolving the DSL is as easy as adding in a new slice [6].

```
module Task {
  role (syntax) {
            Task <-- "task" Identifier "{" StatementList "}"
    StatementList <-- Statement StatementList
    StatementList <-- Statement
  }
  role (evaluation) {
    0 { $2.eval; }
    3 { $4.eval; $5.eval; }
    6 { $7.eval; }
  }
}
slice Task { module Task with role syntax evaluation }
```

Listing 2: Task slice in Neverlang.

```
module Rename {
  role(syntax){
      Rename <-- "rename" Filename Filename
    Statement <-- Rename
  }
  role(evaluation){
    0 { move($1.eval,$2.eval); }
    3 { 4.eval; }
  }
}
slice Rename { module Rename with role syntax evaluation }
```

Listing 3: Rename slice in Neverlang.

We provide the Neverlang code for the `task` construct in Listing 2. `Task` is a `module` that includes two `role`s. The `syntax` role specifies the relevant grammar portion. In this listing, terminals are quoted, and any other symbol is a nonterminal. In particular a `task` has a name (`Identifier`) and it groups together a list of `Statements`. Please notice that, in this slice, neither `Identifier` nor `Statement` are explicitly defined, even though we can reasonably expect that they will be defined in other slices. For instance, Listing 3 shows code for the `rename` statement. Similar slices are defined for `remove` and `backup`. In fact, *unbound* nonterminals such as `Identifier` and `Statement` are allowed in a module, but are supposed to be bound in other slices involved in the composition.

The role `evaluation` executes the statements in sequence. Ordinals in the `evaluation` module correspond to nonterminal symbols in the `syntax` module. Nonterminals are numbered from left to right and from top to bottom, starting from 0. In Listing 2, `Task` corresponds to 0, `Identifier` to 1, and so on. When they introduce a block, ordinals always

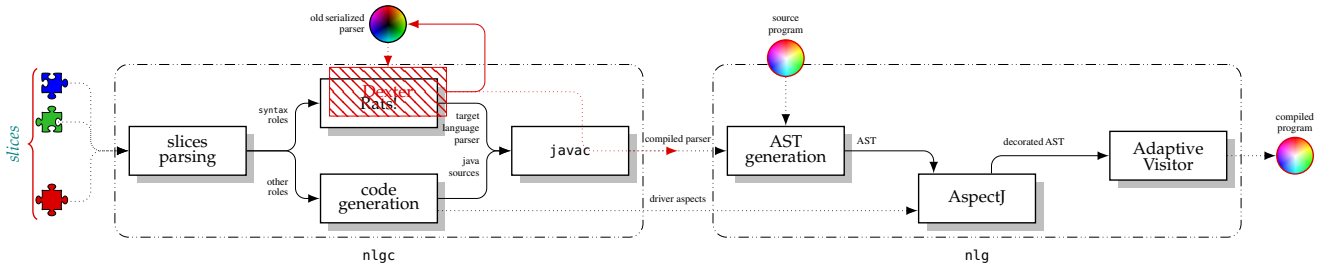

Figure 1: The architecture of Neverlang, with and without DEXTER.

```
language LogLang {
  slices Identifier Main Remove FileSystemOp Logger
         Task Rename BackUp Merge PermissionCheck
}
```

Listing 4: The `language` construct.

correspond to nonterminals on the left-hand side of a rule. For instance, in the `Task` module, a code block introduced by `0` corresponds to the `Task` nonterminal. Nonterminals can be referenced within code blocks, too. In this case, ordinals are prefixed by a dollar sign. For instance, `$1` refers the first `Identifier` symbol from within a code block. In this context, nonterminals are treated as the root of a subtree of the AST of the source file that is being parsed (cf. *syntax-directed translation* for instance in [1]). Each nonterminal has then *attributes* that can be referred using a familiar dot-notation. In the example, we see the attribute `eval`. But there are also other attributes such as `type` and `value`. Accessing `eval` triggers the `evaluation` module of the subtree rooted at that nonterminal, so `$1.eval` does not only return the evaluated content of an `Identifier`, but also executes any code block that is tied to an `Identifier` in some slice.

Slices are finally glued together using the `language` construct (see Listing 4). The Neverlang compiler `nlgc` can be then invoked on the source files.

Under the Hood. The Neverlang compiler `nlgc` (Fig. 1) reads the source slices and generates a compiler or an interpreter for the DSL that is being developed. In this case, `nlgc` composes the interpreter for LogLang. The process consists in a) generating a parser for the new language, b) generating the semantic actions and attaching them to the AST nodes, and finally c) compile everything. The compiler or interpreter can be then executed using the runtime companion `nlg`. Rats! [14] is the parser generator that we chose for Neverlang earliest development stages. Rats! is a parser generator for parsing expression grammars [11]; it is a *traditional* parser generator in that it outputs an intermediate source representation of the parser that must be then compiled. The full input grammar for the target DSL is obtained by extracting and joining the `syntax` portions of every slice involved. When it will be run, the parser's output will be, of course, an AST. Each node of the AST corresponds to a particular grammar symbol. This symbol can be identified among the others with respect to the module, rule and ordinal that identify it in a `syntax` slice. For instance, the `Truncate` nonterminal (see Listing 6) is identified by the `Truncate` module and the ordinal `0`. In order to attach semantic actions to the nodes, this information is encoded by `nlgc` in the type of the AST node. The semantic actions are then encapsulated in aspects that AspectJ weaves into the correct node using type selectors. Finally, `nlg` takes in input Neverlang's compiled files and the source file of the target language. The AST generated by the parser is then traversed by an adaptive visitor. It is easy to see that the compilation phases at this stage was unavoidable: Rats! needs to generate the source files for the parser, and the types for AST nodes have to be generated statically.

Back to our simple example, the LogLang interpreter runs typically as a cronjob, but any user can execute their own LogLang program. Consequently, several instances of the interpreter could be running at a time. Now, imagine that the system administrator observes some misbehavior in the rotating routines. In fact, sometimes it is not possible to request a program to close its log file, thus it might still be appending contents when a `task` starts. The administrator decides to update the LogLang interpreter to support the new `truncate` statement that truncates a log instead of renaming it.

The drawback of the generative approach is that any change in the structure of the DSL might potentially require `nlgc` to regenerate completely the intermediate artifacts, which in turn would require a full recompilation. This is nothing new: if we do not think of the way Neverlang achieves extensibility through composition, implementing non-modular DSLs using traditional tools such as `yacc` and `lex` has always meant to put up with an unavoidable *generate & rebuild* development cycle. In fact, this is often the same routine we have retained to date. Nowadays, modern integrated development environments such as Xtext [9] make the entire process of code generation and recompilation transparent. Modern CPUs are more than capable of performing these intensive tasks, even when builds are frequent, like in the early stages of development. Still, at the end of the day, this feels like sweeping the problem under a carpet.

Our recent efforts are in the direction of minimizing code generation, focusing on the extensibility requirement. We started from rethinking the front-end. The result was the DEXTER library.

Implementing DEXTER. The DEXTER library generates LALR(1) parsers *on the fly*. Traditional parser generators, like `yacc` and even Rats!, are stand-alone tools that take a grammar specification as their input and then statically generate source files for a target language. The generated sources are neither supposed to be modified by a human nor updated by the code generator. Syntax enhancements must be performed on the originating grammar files. In our case, not only has Rats! to regenerate the sources each time, but

it is also necessary that we read again each `syntax` slice to recreate the complete input grammar. We wanted to improve the efficiency of the process. We ended up with taking an entirely different approach to parser generation.

Parsers generated with DEXTER are runtime objects that represent a BNF grammar and the corresponding LALR(1) automaton. LALR(1) [7] is a widely adopted subclass of LR, a class of parsers well-known in literature, due to Knuth [15]. The technique differs from packrat, employed by Rats!. In fact, packrat parsers implement a special linear-time variant of *recursive descent*, which in turn implements a parser of the LL class. The LR technique is attractive for a number or reasons: for instance, the class of recognized grammars includes virtually any programming-language construct that can be described through a context-free grammar, and the algorithm is non-backtracking [1]. An LALR(1) automaton can be represented as a graph, where vertices represent *states* of the parser, and edges correspond to *transitions* between states. The state of the parser at a given time is determined by the input it has been read up to that point. Updating an LALR(1) parser means updating the graph representation.

A DEXTER instance maintains metadata to let developers freely alter the structure of a parser, reflecting changes to the corresponding grammar. Rules can be added and removed, and the parser will *grow* and *shrink* on the fly. Changes are *incremental* in that updates are not performed by re-generating the complete automaton each time. Instead, we implemented algorithms that transform the initial automaton into the updated automaton, by adding to the first only the differences between the two. Of course, the algorithms *do not* actually compute a *delta* each time; otherwise it would be necessary to generate the updated automaton separately, which would defeat the purpose of incremental updates in the first place. Instead, they are actually implemented as a consequence of a series of formal proofs[2], that guarantee that, given a batch of changes to the grammar, only a definite sequence of steps is required to accomplish the update.

A DEXTER object can live and die in the context of a program execution, or it can survive. In this case the instance is serialized to disk, so it can be reloaded back to parse a source input, or to perform new updates. This is a significant departure from the previous code generation approach, because the generated artifact is a serialized object, and not just source code requiring compilation.

3. DEXTER AND NEVERLANG

In this section we describe the necessary steps to integrate DEXTER with Neverlang. The approach to parser generation taken in DEXTER requires a few changes in the way the framework works. This is also a good occasion to take a step back, look overall at the current architecture, and possibly make breaks with the past to bring Neverlang further.

DEXTER's APIs. A DEXTER parser is an instance of the `Dexter` class. This class principally provides the primitives to *grow* the parser with new rules (`add`) and to *shrink* the parser by deleting old rules (`remove`). As you would expect, the `parse` method returns a parse tree when parsing succeeds, or an error otherwise. There is also a straightforward API to store and load the generated parser to a file, that is implemented using Java's native serialization features. The serialized form of a parser generated using DEXTER retains

[2]We plan to publish these results in another work

```
Dexter dx = new Dexter();

dx.add(Task, "task", Identifier, "{", StatementList, "}");
dx.add(StatementList, Statement, StatementList);
dx.add(StatementList, Statement);

dx.add(Rename, "rename", Filename, Filename);
dx.add(Statement, Rename);
```

Listing 5: DEXTER API calls for LogLang.

```
module Truncate {
  role(syntax) {
    Truncate <— "truncate" Identifier
    Statement <— Truncate
  }
  role(evaluation) {
    0 { /* truncate $1.eval */ }
    2 { $3.eval; }
  }
}
slice Truncate { module Truncate with role syntax evaluation }
```

Listing 6: The `Truncate` statement.

all the metadata that is necessary to perform updates. In other words, a DEXTER-generated parser can be freely updated even when it has been loaded back from a file. Listing 5 shows the relevant API calls for LogLang's `syntax` slices that we showed earlier.

Neverlang's workflow favors code reuse in that an existing slice can be imported into another DSL by just including it in its correspondent `language` construct (Listing 4). When invoked on the new DSL, the Neverlang compiler `nlgc` (see Fig. 1) processes every slice, old and new, and generates the intermediate artifacts (grammar input files for the parser generator, Java and AspectJ source files to implement the semantic actions) that we previously described in Sect. 2. The ability to share slices across different DSLs is an effective way to promote modularity, but, unfortunately, the intermediate code generation step is always unavoidable. This, in the case of old slices, is suboptimal. In particular, at the end of Sect. 2, we supposed that a system administrator wanted to extend LogLang with a `truncate` statement. The system administrator of our example chooses the syntax of the command to look as follows:

```
truncate filename
```

In order to produce the new interpreter, the statement is defined in the slice of Listing 6, and the `language` statement is updated accordingly. Finally, `nlgc` is invoked on the slices, and a new interpreter is produced.

DEXTER makes possible to *incrementally* update the parser. In this case, the old serialized parser is updated by `nlgc` by loading the serialized DEXTER object and adding the new rules. In our example, adding the productions for the `truncate` statements requires only four API calls, two to load the `Dexter` object from file and save it back and the others to actually add the productions (see Listing 7). Using DEXTER, the Neverlang compiler can generate a parser by adding or removing the production rules contained in a `syntax` role *incrementally*. In other words, an update to a DSL developed with Neverlang, like the one that we imagined for the `Truncate` slice in LogLang, would only require `nlgc` to perform an incremental change to the serialized parser.

Pursuing Integration. We already said that DEXTER is meant to substitute Rats! in `nlgc` (Fig. 1), but, due to its dual nature of parser generator and runtime object, it is

```
Dexter dx = Dexter.load(serializedParserFile);
dx.add(Truncate, "truncate", Filename);
dx.add(Statement, Truncate);
dx.save(serializedParserFile);
```

Listing 7: API calls to add the `truncate` statement.

not just a drop-in replacement. First, there is a gap between DEXTER's internal representation of a grammar and Neverlang's, that involves how `nlgc` uses DEXTER; but, most of all, there are differences between Neverlang's representation of a syntax tree and DEXTER's. These differences makes AspectJ's role more prominent in the runtime part (`nlg`) of Neverlang.

As a parser generator, we saw that DEXTER knows about the concept of syntactic rule, but in fact, it is completely unaware of the notions of slice and module. A `syntax` role in Neverlang generally contains more than one single production, but rather it contains several related rules, so we would like these rules to maintain their relation. The void can be easily filled by extending DEXTER's API. The API calls shown in Listing 5 and 7 are really shorthands for a longer form. For instance, with respect to the first production in the `Rename` slice, the API call is internally translated to the following method call:

```
dx.add(new Production(Rename, "rename", Filename, Filename));
```

Metadata about slices and modules can be attached to rules in a separate internal table using the extended form:

```
dx.add(new Production(Rename, "rename", Filename, Filename),
       new NeverlangMetaData(moduleId, sliceId));
```

By the help of this data structure, the API can be further enhanced, making possible to remove productions on a per-slice basis.

```
dx.remove(moduleId,sliceId);
```

There is still one point missing. A Neverlang-generated interpreter (or compiler) executes code using the *syntax-directed translation* technique [1]. The input file is translated into an AST, and then code is executed as the tree is visited. Rule-specific executable code, described in a slice, is injected as a method of the corresponding AST node using AspectJ. Before DEXTER, Neverlang's AST nodes always mapped to one and only one symbol, in a specific rule of a given `syntax` role. This information was directly encoded in the type name of a subclass of a common `ASTNode` type. It was then possible to inject code using a type selector. This simple mapping was possible at the time because of the completely generative nature of Neverlang. The internal toolchain could make use of these new types, because everything was generated statically.

DEXTER's syntax tree does not encode information about symbols in the type of a node. DEXTER's `ASTNodes` include a `symbol` field and a `production` field, instead. The `symbol` field represents a terminal if the node is a leaf, or a nonterminal otherwise. In the case of a terminal, the symbol represents a string of the recognized input. Otherwise, the `production` field represents the grammar rule whose left-hand part corresponds to `symbol`. For instance, recalling our example, in LogLang an `ASTNode`'s `production` field could contain `Statement ⇐ Rename` or `Statement ⇐ Truncate` when the `symbol` field is `Statement`. This is already enough information to distinguish nodes. However, instead of the simple type selectors that can instrument classes statically,

```
class TruncateModule extends neverlang.Module {
 @Role SyntaxRole syntax = new SyntaxRole() {
   void syntax(Dexter dx) {
     /* rule 0 */
     dx.add(new Production(Truncate, "truncate", Identifier),
            getNeverlangMetadata());
     /* rule 1 */
     dx.add(new Production(Statement, Truncate),
            getNeverlangMetadata());
   }
 };
 @Role EvaluationRole evaluation = new EvaluationRole() {
   @BindRule(0)
   void rule0 (ASTNode n) { /* truncate n.child[1] */ }
   @BindRule(1)
   void rule1 (ASTNode n) { n.child[0].eval(); }
 };
}
class SliceTruncate extends neverlang.Slice {
 public SliceTruncate() {
  this.modules = new Modules[] { new TruncateModule() };
 }
}
```

Listing 8: Neverlang modules as Java objects.

Neverlang must employ more powerful AspectJ pointcut constructs that evaluate dynamic properties (namely, the `if` pointcut expression). The result is less generated code, at the cost of dynamic weaving (see Fig 1). In fact, this is actually an advantage, rather than a limitation, because the final goal for Neverlang is full language extensibility at runtime, at the cost of a small loss in performances, due to dynamic weaving.

Performances. Of course, it is reasonable to expect that incremental updates to an existing parser take less time than regenerating it anew using traditional parser generators, and our early results in this field are very promising. On a test machine, a Windows 7 x64 PC with a Core 2 Duo P8400, 2.26GHz per core, 4GB of RAM, and JDK 6, any call to the `add` or `remove` methods in general does not take more than 1/100 second. Initializing the C grammar takes always less than 2 seconds, 1.5 on average. Considering that the implemented C89 grammar contains 220 productions, we are quite satisfied of the result. As for parsing performances, they vary depending on the grammar, but for our intents, they have been proven to be reasonable. For instance, parsing a 100-lines C file takes 55 milliseconds on average.

The Road to Runtime Extensibility. Even though at the moment code generation is still necessary, with DEXTER the way is now open to move Neverlang forward to a completely dynamically extensible language generation system. Our final objective is achieve full runtime extensibility.

We already mentioned integrated development environments for language implementation as one possible application. Martin Fowler's essay [12] calls these environments a *killer application* for a more mainstream adoption of language-oriented approaches to problem solving. The whole idea of language-oriented programming [19], is dear to Unix administrators and Lisp developers. The former is reflected in the myriad of *little languages* [3] implemented in Unix tools like `awk` or `make`. The latter is because of the metasyntactic capabilities that render LISP, according to [10], a *programmable programming language*. But Fowler's recent essay builds upon the concept, and stresses that IDE support is very important. Yet, today, IDEs for language development are still constrained to the same old *generate & rebuild* cycle that we want to avoid.

Neverlang as a library might be the answer to the need for more flexible and efficient IDEs. The DEXTER parser is a move in the right direction, but there is still some work to do, in order to avoid or dramatically minimize intermediate code generation steps.

If we want to reach this goal, we need first to implement a dynamic lexer solution to tokenize the input stream. In fact, at the moment, DEXTER still relies on a traditional code generator for the lexing part. The lexer is generated by ANTLR [18]. Lexing and parsing are now decoupled, but we are considering solutions to implement a dynamically extensible lexer, integrated with DEXTER, and completely independent from ANTLR.

The second problem is that currently aspects are woven in an offline compilation step. Projects such as CaesarJ [2] support dynamic deployment and undeployment of aspects, and this is a direction that is worth investigating. The weaver's current prominent role could be even reconsidered and possibly cut down to that of *system advisor* that only forwards method calls to objects encapsulating a behavior. For instance, a subclass of a `Role` class could encapsulate the actions in a given role, and subclasses of `Module` would collect slices together. Precompiled slices and modules would not need to be regenerated unless their correspondent Neverlang source file changed: binary modules and slices could be shared across DSLs and loaded dynamically by Neverlang's advisor. Listing 8 shows the possible Java source of a binary representation of the `Truncate` module in Listing 6. The intermediate source representation could be avoided by directly compiling Neverlang source files to bytecode. In the example, there is a simple mapping between the concepts of slice, module and role to Java native data structures. A role becomes an annotated field of the `Module` class. A `syntax` role is encapsulated in a method of the interface `SyntaxRole` that operates on a `Dexter` instance. Finally, any other role implements a subinterface of the interface `Role`. In the example, the `RoleEvaluation` interface. `Role` interfaces act as containers for void methods that take an `ASTNode` as argument. They are bound to the corresponding rule in the `syntax` role using a number. The first rule in `syntax` is rule `0`, the second is rule `1` and so on. For instance, the method `rule0` is logically bound to the rule `Truncate ⇐ "truncate" Identifier`. Then, at runtime, the Neverlang advisor could execute the method `rule0()` each time a corresponding `ASTNode` is visited during the evaluation phase. When the method is called, `ASTNode n` would contain the `Truncate` symbol in its `symbol` field, and that rule in the `production` field. The `n.child` array contains the children of the tree rooted at `Truncate`. For instance, `n.child[1]`.

4. CONCLUSIONS AND FUTURE WORKS

In this paper we presented the DEXTER parser generator library and the process of integration with the Neverlang framework. Our earlier results are promising and make us confident to continue work on Neverlang. In this paper we described how DEXTER fits in the Neverlang framework. We will be experimenting with Neverlang's syntax to extend it and enhance it. Moreover, we must never forget Martin Fowler's considerations [12]: the importance of great tooling is something that today we cannot ignore for the adoption of a technology. There is still a lot of work to do in this direction. For instance, integrated debugging would help to implement complex grammars.

In the last part we also described how moving from a static generative process to a more dynamic environment can push the boundaries of Neverlang further. Moving to a dynamic approach can lead the way for new interesting developments. The world of integrated development environments is only one possible application for the Neverlang framework. In general, any long-running application could potentially benefit from hot deployment of new features, and we are therefore even more interested to head Neverlang's future development in this direction.

Acknowledgments

This work is part of the DISCO project funded by Italian MIUR.

5. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *TAOSD*, 1(1):135–173, 2006.

[3] J. Bentley. Programming Pearls: Little Languages. *Commun. ACM*, 29(8):711–721, Aug. 1986.

[4] J. Bovet and T. J. Parr. ANTLRWorks: an ANTLR Grammar Development Environment. *Software—Practice and Experience*, 38(12):1305–1332, 2008.

[5] W. Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proc. of SC'12*, LNCS 7306, pp. 162–177, Prague, Czech Republic, June 2012.

[6] W. Cazzola and D. Poletti. DSL Evolution through Composition. In *Proc. of RAM-SE'10*, Maribor, Slovenia, June 2010. ACM.

[7] F. DeRemer and T. J. Pennello. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Trans. Prog. Lang. Syst.*, 4(4):615–649, Oct. 1982.

[8] E. W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, Oct. 1972.

[9] S. Efftinge and M. Völter. oAW xText: A Framework for Textual DSLs. In *Proceedings of ESE'06*, Nov. 2006.

[10] J. Foderaro. LISP: Introduction. *Commun. ACM*, 34(9):27, Sept. 1991. Special Issue on Lisp.

[11] B. Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In *Proc. of POPL'04*, pp. 111–122, Venice, Italy, Jan. 2004. ACM.

[12] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Fowler's Blog, May 2005.

[13] D. Ghosh. *DSLs in Action*. Manning, Dec. 2010.

[14] R. Grimm. Practical Packrat Parsing. TR2004-854, New York University, New York, NY, USA, 2004.

[15] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[16] F. Marguerie, S. Eichert, and J. Wooley. *LINQ in Action*, chapter "Introducing LINQ". Manning, 2008.

[17] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Aritma Press, 2008.

[18] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[19] M. P. Ward. Language Oriented Programming. *Software - Concept and Tools*, 15(4):147–161, 1994.

[20] A. Watters, G. van Rossum, and J. C. Ahlstrom. *Internet Programming with Python*. MIS Press, Sept. 1996.