A MODULAR ARCHITECTURE FOR CONSTRAINT-BASED PARSING

François Barthélemy[◇♣] François Rouaix[◇]

◇ INRIA Rocquencourt, BP 105, 78153 Le Chesnay cedex, France
♣ Universidade Nova de Lisboa, 2825 Monte de Caparica, Portugal

Abstract

This paper presents a framework and a system for implementing, comparing and analyzing parsers for some classes of Constraint-Based Grammars. The framework consists in a uniform theoretic description of parsing algorithms, and provides the structure for decomposing the system into logical components, with possibly several interchangeable implementations. Many parsing algorithms can be obtained by composition of the modules of our system. Modularity is also a way of achieving code sharing for the common parts of these various algorithms. Furthermore, the design helps reusing the existing modules when implementing other algorithms. The system uses the flexible modularity provided by the programming languages Alcool–90, based on a type system that ensures the safety of module composition.

1 Introduction

We designed a system to study parsing. Our aim was not to implement only one parsing algorithm, but as many as possible, in such a way that we could compare their performances. We wanted to study parsers' behavior rather than using them to exploit their parses. Furthermore, we wanted a system opened to new developments, impossible to predict at the time we began our project.

We achieved these aims by defining a modular architecture that gives us in addition code sharing between alternative implementations.

Our system, called APOC–II, implements more than 60 different parsing algorithms for Context-Free Grammars, Tree-Adjoining Grammars, and Definite-Clause Grammars. The different generated parsers are comparable, because they are implemented in the same way, with common data structures. Experimental comparison can involve more than 20 parsers for a given grammar and give results independent from the implementation.

Furthermore, adding new modules multiplies the number of parsing algorithm. APOC–II is open to new parsing techniques to such an extent that it can be seen as a library of tools for parsing, including constraint solvers, look-ahead, parsing strategies and control strategies. These tools make prototyping of parsing algorithms easier and quicker.

The system is based on a general framework that divides parsing matters in three different tasks. First, the compilation that translates a grammar into a push-down automaton describing how a parse-tree is built. The automaton can be non-deterministic if several trees have to be considered when parsing a string. Second, the interpretation of the push-down automaton that has to deal with non-determinism. Third, the constraint solving, used by both compilation and interpretation to perform operations related to constraints.

Several algorithms can perform each of these three tasks: the compiler can generate either top-down or bottom-up automata, the interpreter can make use of backtracking or of tabulation and the solver has to deal with different kinds of constraints (first-order terms, features, ...).

Our architecture allows different combinations of three components (one for each basic task) resulting into a specific parsing system. We use the Alcool–90 programming language to implement our modules. This language's type system allows the definition of alternative implementations of a component and ensures the safety of module combination, i.e. each module provides what is needed by other modules and receives what it requires.

The same kind of modularity is used to split the main components (compiler, interpreter, solver) into independent sub-modules. Some of these sub-modules can be shared by several different implementations. For instance the computation of look-ahead is the same for LL(k) and LR(k) techniques.

The next section defines the class of grammar we consider. Then, a general framework for parsing and the sort of modularity it requires are presented. Section 4 is devoted to the Alcool–90 language that provides a convenient module system. Section 5 is the detailed description of the APOC–

II system that implements the general framework using Alcool–90.

2 Constraint-Based Grammars

The notion of Constraint-Based Grammar appeared in computational linguistic. It is a useful abstraction of several classes of grammars, including the most commonly used to describe Natural Language in view of computer processing.

We give our own definition of constraint-based grammars that may slightly differ from other definitions.

Definition 1 *Constraint-Based Grammar*
A constraint-based grammar is a 7-tuple $\{Nt, T, \alpha, V, Ax, CL, R\}$ *where*

- *Nt is a set of symbols called non-terminals*
- *T is a set of symbols called terminals*
- *α is a function from $Nt \cup T$ to the natural integers called the arity of the symbols*
- *V is an infinite set of variables*
- *Ax is an element of Nt called the axiom*
- *CL is a constraint language (see definition below) having V as variable set and being closed under renaming and conjunction*
- *R is a finite set of rules of the form:*

$$s_0(\vec{X_0}) \rightarrow c, s_1(\vec{X_1}), \ldots, s_n(\vec{X_n}).$$

such that $s_0 \in Nt, s_i \in Nt \cup T$ for $0 < i \leq n$, $c \in CL$, $\vec{X_i}$ are tuples of $\alpha(s_i)$ distinct variables, and the same variable cannot appear in two different tuples.

In this definition, we use the notion of *constraint language* to define the syntax and the semantics of the constraints used by the grammars. We refer to the definition given by Höfeld and Smolka in [HS88]. This definition is especially suitable for constraints used in NLP (unrestricted syntax, multiplicity of interpretation domains). The *closure under renaming* property has also been defined by Höfeld and Smolka. It ensures that constraints are independent from the variable names. This grounds the systematic renaming of grammar rules to avoid variable conflicts.

Definition 2 *Constraint Language*
A constraint Language is a 4-tuple (V,C,ν,I) such that:

- *V is an infinite set of variables*
- *C is a decidable set whose elements are called constraints*
- *ν is function that associates a finite set of variables to each constraint*
- *I is a non-empty set of interpretations*

For lack of space we do not recall in detail what an interpretation and the "closure under renaming" property are, and refer to [HS88].

The semantics of Constraint-Based Grammars is defined by the semantics of the constraint language and the notion of syntax tree. A syntax tree is a tree which has a grammar rule (renamed with fresh variables) as label of each node. A constraint is associated to a parse tree: it is the conjunction of all the constraints of the labels and the equalities between the tuple of variables from the non-terminal of the left-hand side of a label and the tuple of the relevant symbol of the right-hand side of the label of its parent.

An important point about parse trees is that the order of terminal symbols of the input string and the order of the symbols in right-hand sides of rules are significant.

A Context-Free Grammar is obtained just by removing tuples and constraints from the grammar rules. Most parsing techniques for Constraint-Based Grammars use the underlying context-free structure to guide parsing. This allows the reuse of context-free parsing techniques.

The grammars we have just defined encompass several classes of grammars used in NLP, including logic grammars (Definite Clause Grammars and variants), Unification Grammars, Tree Adjoining Grammars[1] and, at least partially, Lexical-Functional Grammars and Head Phrase Structure Grammars. Of course, there are syntactical differences between these classes and Constraint-Based Grammars. A simple translation from one syntax to the other is necessary.

3 A General Framework for Parsing

This section is devoted to a general framework for parsing in which most of the parsing methods, including all the most common ones, are expressible. It is an extension of a context-free framework [Lan74]. It is based on an explicit separation between the *parsing strategy* that describes how

[1]TAGs have an underlying context-free structure, although this is not obvious in their formal definition. See for instance [Lan91].

syntax trees are built (e.g. top-down, bottom-up), and the *control strategy* that deals with the non-determinism of the parsing (e.g. backtracking, tabulation).

3.1 EPDAs

This separation is based on an intermediate representation that describes how a grammar is used following a given parsing strategy. This intermediate representation is a Push-Down Automaton. It is known that most context-free parsers can be encoded with such a stack machine. Of course, the usual formalism has to be extended to take constraints into account, and possibly use them to disambiguate the parsing. We call Extended Push-Down Automaton (EPDA) the extended formalism.

For lack of space, we do not give here the formal definition of EPDA. Informally, it is a machine using three data structures: a stack containing at each level a stack symbol and its tuple of variables; a representation of the terminal string that distinguishes those that have already been used and those that are still to be read; finally a constraint. A configuration of an automaton is a triple of these three data. Transitions are partial functions from configurations to configurations. We add some restrictions to these transitions: the only change allowed for the string is that at most one more terminal is read; only the top of the stack is accessible and at most one symbol can be added or removed from it at once. These restrictions are needed to employ directly the generic tabular techniques for automata execution described in [BVdlC92]. EPDAs may be non-deterministic, i.e. several transitions are applicable on a given configuration.

Parsing for Constraint-Based Grammars blends two tasks:

- The structural part, that consists in building the skeleton of parse trees. This part is similar to a context-free parsing with the underlying context-free projection of the grammar.
- Solving the constraints of this skeleton.

The two tasks are related in the following way: constraints appear at the nodes of the tree; the structure is not a valid syntax tree if the constraint set is unsatisfiable. Each task can be performed in several ways: there are several context-free parsing methods (e.g. LL, LR) and constraints sets can be solved globally or incrementally, using various orders, and several ways of mixing the two tasks are valid. Tree construction involves a stack mechanism, and constraint solving results in a constraint. The different parsing techniques can be described as computations on these two data structures. EPDAs are thus able to encode various parsers for Constraint Grammars.

Automatic translation of grammars into EPDAs is possible using extensions of usual context-free techniques [Bar93].

3.2 Architecture

Thanks to the intermediate representation (EPDA), parsing can be divided into two independent passes: the compilation that translates a grammar into an extended automaton; the execution that takes an EPDA and a string and produces a forest of syntax trees. To achieve the independence, the compiler is not allowed to make any assumptions about the way the automata it produces will be executed, and the interpreter in charge of the execution is not allowed to make assumptions about the automata it executes.

We add to this scheme reused from context-free parsing a third component: the solver (in an extensive meaning) in charge of all the operations related to constraints and variables. We will try to make it as independent from the other two modules (compiler and interpreter) as possible.

There is not a full independence, since both the compiler and the interpreter involve constraints and related operations, that are performed by the solver. We just want to define a clear interface between the solver and the other modules, an interface independent from the kind of the constraints and from the solving algorithms being used. The same compiler (resp. interpreter) used with different solvers will work on different classes of grammars. For instance, the same compiler can compile Unification Grammars and Definite Clause Grammars, using two solvers, one implementing feature unification, the second one implementing first-order unification.

We can see a complete parsing system as the combination of three modules, compiler, interpreter, solver. When each module has several implementations, we would like to take any combination of three modules. This schematic abstraction captures parsing algorithms we are interested in. However, actually defining interfaces for a practical system without restricting open-endedness or the abstraction (interchangeability of components) was the most difficult technical task of this work.

3.3 Solvers

The main problem lies in the definition of the solver's interface. Some of the required operations are obvious: renaming of constraints and tuples, constraint building, extraction of the variables from a constraint, etc.

By the way, remark that constraint solving can be hidden within the solver, and thus not appear in the interface. There is an equivalence relation between constraints given by their interpretations. This relation can be used to replace a constraint by another equivalent one, possibly simpler. The solving can also be explicitly used to enforce the simplification of constraints at some points of the parsing.

Unfortunately some special techniques require more specific operations on constraints. For instance, a family of parsing strategies related to Earley's algorithm make use of the *restriction* operator defined by Shieber in [Shi85]. Another example: some tabular techniques take benefit from a projection operator that restricts constraints with respect to a subset of their variables.

We could define the solver's interface as the cartesian product of all the operations used by at least one technique. There are two reasons to reject such an approach. The first one is that some seldom used operations are difficult to define on some constraints domains. It is the case, among others, of the projection. The second reason is that it would restrict to the techniques already existing and known by us at the moment when we design the interface. This contradicts the open-endedness requirement. A new operation can appear, useful for a new parsing method or for optimizing the old ones.

We prefer a flexible definition of the interface. Instead of defining one single interface, we will allow each alternative implementation of the solver to define exactly what it offers and each implementation of the compiler or of the interpreter to define what it demands. The combination of modules will involve the checking that the *offer* encompasses *the demand*, that all the needed operations are implemented. This imposes restrictions on the combination of modules: it is the overhead to obtain an open-ended system, opened to new developments.

We found a language providing the kind of flexible modularity we needed: Alcool–90. We now present this language.

4 The language Alcool 90

Alcool–90 is an experimental extension of the functional language ML with run-time overloading [Rou90]. Overloading is used as a tool for seamless integration of abstract data types in the ML type system, retaining strong typing, and type inference properties. Abstract data types (encapsulating a data structure representation and its constructors and interpretive functions) provide values for overloaded symbols, as classes provide methods for messages in object-oriented terminology. However, strong typing means that the compiler guarantees that errors of kind "method not found" never happen.

Abstract programs are programs referring to overloaded symbols, which values will be determined at run-time, consistently with the calling environment. By grouping abstract programs, we obtain parameterized abstract data types (or functors), the calling environment being here a particular instantiation of the parameterized adt. Thus, we obtain an environment equivalent to a module system, each module being an adt, eventually parameterized.

For instance, in APOC–II, compilers have an abstract data type parameterized by a solver.

Alcool–90 also proposes an innovative environment where we exploit ambiguities due to overloading for semi-automated program configuration : the type inference computes interfaces of "missing" components to complete a program, according to the use of overloaded symbols in the program. A search algorithm finds components satisfying those interfaces, eventually by finding suitable parameters for parameterized components. Naturally, instantiation of parameterized components is also type-safe : actual parameters must have interfaces matching formal parameters (schematically : the actual parameter must provide at least the functions required by the interface of the formal parameter).

For instance, only the solvers providing Shieber's restriction can be used as the actual parameter of Earley with restriction compiler. But these solvers can also be used by all the compilers that do not use the restriction.

Simple module systems have severe limitations when several implementations of components with similar interfaces coexist in a system, or when some component may be employed in different contexts. Ada generics provided a first step to module parameterization, though at the cost of heavy declarations and difficulties with type equivalence. SML proposes a very powerful module system with parameterization, but lacks separate compilation and still requires a large amount of user declarations to define and use functors. Object-oriented languages lack the type security that Alcool–90 guarantees.

The Alcool–90 approach benefits from the simplification of modules as abstract data types by adding inference facilities: the compiler is able to infer the interfaces of parameters required by a module. Moreover, the instantiation of a functor is simply seen as a type application, thus no efforts are required from the programmer, while its consistency is checked by the compiler.

This approach is mostly useful when multiple implementations with similar interfaces are available, whether they will coexist in the program or they will be used to generate several configurations. Components may have similar interfaces but different semantics, although they are interchangeable. Choosing a configuration is simply choosing from a set of solutions to missing components, computed by the compiler.

Several other features of Alcool–90 have not been used in this experiment, namely the inheritance operator on abstract data types, and an extension of the type system with dynamics (where some type checking occurs at run-time).

5 APOC-II

APOC–II is a system written in Alcool–90, implementing numerous parsing techniques within the framework described in section 3. The user can choose between these techniques to build a parser. By adding new modules written in Alcool–90 to the library, new techniques can freely be added to the system.

APOC–II has two levels of modularity: the first one is that of the three main components distinguished above, compiler, interpreter and solver. Each of these components is implemented by several alternative modules, that are combinable using Alcool–90 discipline.

The second level of modularity consist in splitting each of the three main components into several modules. This makes the sharing of common parts of different implementations possible.

We give now examples of splitting APOC–II uses at the moment, in order to give an idea of this second level of modularity. This splitting has proved convenient so far, but it is not fixed and imposed to further developments: a new implementation can be added even if it uses a completely different internal structure.

A solver is made of:

- a module for variables, variable generation and renaming,
- a parser for constraints,
- a pretty-printer for constraints,
- a constraint builder (creation of abstract syntax trees for constraints, e.g. building constraints expressing equality of variables),
- a solver in the restrictive meaning, in charge of constraint reduction,
- an interface that encapsulate all the other modules.

A compiler includes:

- a grammar parser (that uses the constraint parser given by the solver),
- a module for look-ahead (for computation of look-ahead sets by static analysis of the grammar),
- a module for EPDA representation and handling,
- a transition generator which translates grammar rules into EPDA transitions therefore determining the parsing strategy (cf. figure 1),
- Control code, using previous modules, defining the "compile" function, the only one exported.

The two interpreters implemented so far have very different structures. The first one uses backtracking and the second one uses tabulation. They share some modules however, such as a module handling transitions and a lexer of input strings.

The interest of the modular architecture is in the combinatorial effect of module composition. It leads to many different parsing algorithms. The figure 1 summarizes the different aspects of the parsing algorithms that can vary more or less independently.

For example, the built-in parsing method of Prolog for DCGs is obtained by combining the solver for DCGs, the top-down strategy, 0 symbol of look-ahead and a backtracking interpreter (and other modules not mentioned in figure 1 because they do not change the algorithm, but at most its implementation).

Some remarks about figure 1:

- we call *Earley parsing strategy* the way Earley deduction [PW83] builds a tree, not the control method it uses. It differs from top-down by the way constraints are taken into account.
- the difference between Earley-like tabulation and graph-structure stacks is the data structure used for item storage. Several variants are possible, that actually change the parser's behavior.

Solver (grammar class)	**Context-free Grammars - Definite Clause Grammars Tree Adjoining Grammars** - *Unification Grammars ...*
parsing strategy (transition generator)	**top-down - pure bottom-up - Earley - Earley with restriction left-corner - LR** - *precedence - PLR ...*
look-ahead	**context-free look-ahead of 0 or 1 symbol** *context-free look-ahead of k symbols - context-sensitive look-ahead*
interpreter	**backtracking - Earley-like tabulation** - *Graph-structured Stacks ...*
Agenda management (for tabulation only)	**Synchronization** - *lifo - fifo - various weights ...*

Figure 1: modules of APOC–II
Modules written in bold font are already implemented, whereas modules written in italic are possible extensions to the system.

- we call synchronization a kind of breadth-first search where scanning a terminal is performed only when it is needed by all the paths of the search-tree. The search is synchronized with the input string. It is the order used by Earley's algorithm.
- at the moment, only *generic* look-ahead, that is look-ahead based on the *first* and *follow* sets, has been considered. Some more accurate look-ahead techniques such as the ones involved in SLR(k) parsing are probably not independent from the parsing strategy and cannot be an independent module.

Building a parsing system with APOC–II consists roughly in choosing one module of each row of figure 1 and combining them. Some of the combinations are not possible. Thanks to type-checking, Alcool–90 will detect the incompatibility and provide a type-based explanation of the problem.

At the moment, APOC–II offers more than 60 different parsing algorithms. Given a grammar, there is a choice of more than 20 different parsers. Adding one module does not add only one more algorithm, but several new variants.

The techniques implemented by APOC–II are not original. For instance, the LR compilation strategy comes from a paper by Nilsson, [Nil86], left-corner parsing has been used by Matsumoto and Tanaka in [MT83]. As far as we know, however, LR and left-corner parsers have not been proposed for Tree-Adjoining Grammars before.

Notice that the modularity is also useful to vary implementation of algorithms. For instance, a first prototype can be quickly written by implementing constraints reduction in a naive way. A refined version can be written later, if needed.

6 Conclusion

APOC–II has several advantages. First of all, it provides comparable implementations of the most common parsing algorithms. Their efficiency can be abstractly measured, for instance by counting the number of computation step (EPDA transition application) performed to compute a tree or a complete forest of parse trees. We call this kind of measurements abstract because it does not rely neither on the implementation nor on the machine that runs the parser. Other comparisons could be done statically, on the automaton or on the parse forest (e.g. number of transitions, amount of determinism, size of the forest, amount of structure sharing).

Otherwise, APOC–II can be used as a toolkit that provides a library of modules useful to implement quickly new parser generators. For instance, one has only to write a solver to obtain up to 22 parsing algorithms (perhaps less if the solver provides only basic operations). The library contains tools to deal with some constraints, look-ahead, lexing, tabulation, etc. Reusing these tools whenever it is possible saves a lot of work.

The limitations of APOC–II are that it is mainly convenient for parsing strategies that are somehow *static*, i.e. statically determined at compile time. Also, abstraction (full independence between compilers and interpreters) cannot be achieved for some optimized algorithms. For instance, Nederhof presents in [Ned93] a parsing strategy called ELR for which tabular execution can be optimized. To implement this algorithm in our system, one would have to write a new interpreter dedicated to ELR-EPDAs.

We think that our experiment shows the interest of a flexible modularity for studies about parsing. We believe that the same technique can fruitfully apply on other domains of Natural Language Processing.

7 Acknowledgements

The authors are grateful to Gabriel Pereira Lopes for his help.

References

[Bar93] François Barthélemy. Outils pour l'analyse syntaxique contextuelle. Thèse de doctorat, Université d'Orléans, 1993.

[BVdlC92] F. Barthélemy and E. Villemonte de la Clergerie. Subsumption-oriented push-down automata. In *Proc. of PLILP'92*, pages 100–114, june 1992.

[HS88] M. Höhfeld and G. Smolka. Definite Relations over Constraint Languages. Technical Report 53, LILOG, IWBS, IBM Deutschland, october 1988.

[Lan74] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *Proc. of the 2^{nd} Colloquium on automata, languages and Programming*, pages 255–269, Saarbrücken (Germany), 1974. Springer-Verlag (LNCS 14).

[Lan91] Bernard Lang. The systematic construction of earley parsers: Application to the production of o(n^6) earley parsers for tree adjoining grammars. In *First International Workshop on Tree Adjoining Grammars*, 1991.

[MT83] Y. Matsumoto and H. Tanaka. Bup: A bottom-up parser embedded in prolog. *New Generation Computing*, 1:145–158, 1983.

[Ned93] Mark-Jan Nederhof. A multidisciplinary approach to a parsing algorithm. In *Proceedings of the Twente Workshop on Language Technology - TWLT6*, december 1993.

[Nil86] Ulf Nilsson. Aid: An alternative implementation of DCGs. *New Generation Computing*, 4:383–399, 1986.

[PW83] F. C. N. Pereira and D. H. D. Warren. Parsing as deduction. In *Proc. of the 21st Annual Meeting of the Association for Computationnal Linguistic*, pages 137–144, Cambridge (Massachussetts), 1983.

[Rou90] François Rouaix. ALCOOL-90: Typage de la surcharge dans un langage fonctionnel. Thèse de doctorat, Université Paris 7, 1990.

[Shi85] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23^{rd} Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago (Illinois), 1985.