This paper appears in the *Proceedings of the Intl. Workshop on Attribute Grammars and their Applications*, ed. by P. Deransart and M. Jourdan, Paris (F), September 19–21, 1990, pp. 222-237, vol. 461 of *Lect. Notes in Comp. Sci.*, Springer-Verlag. Copyright © 1990 by Springer-Verlag, New-York, Heidelberg, Berlin.

The OLGA Attribute Grammar Description Language: Design, Implementation and Evaluation

Martin JOURDAN, Carole LE BELLEC and Didier PARIGOT
INRIA*

Abstract

OLGA is the input language of the FNC-2 attribute grammar processing system, currently under development at INRIA. As such, it is designed for the specification of attribute grammars and is specialized for this purpose. The features of OLGA can be classified into those which make it a powerful general-purpose applicative language and those which make it a specialized AG-description language. A remarkable feature of OLGA is its strong support for modularity. The paper discusses the design goals for OLGA and presents the most important aspects of the language. It also includes comparisons with other existing languages, an overview of the implementation of OLGA, namely the FNC-2 system, and an account of the experience gained in using OLGA.

1 Introduction and Design Goals

Since attribute grammars (AGs) have been introduced in Knuth's seminal paper [Knu68], they have become a method of choice for describing and implementing syntax-directed computations, in particular compilers, translators and syntax-directed editors. Many people have made much research work on various aspects of AGs, and many systems have been developed around the world which produce efficient attributes evaluators [DJL88]. However, surprisingly few of these works dealt with the design of languages suited to AG writing. Indeed, most systems use as input language a variant of their implementation language merely augmented with constructs for attributes computation and access.

Our experience in using the FNC/ERN system and Lisp for developing large applications, and that of the OPTRAN team with their Pascal-based system [LMW88, sect. 4], uncovered many flaws with this approach:

- The AG framework requires that the semantic rules be written in a purely applicative fashion because, when side effects are involved, the order of their execution cannot be guaranteed. Most implementation languages provide imperative constructs (e.g. assignment to some global variable) which cannot be controlled by the AG-processing system.

*Authors' address: INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex (France). E-mail: {jourdan,lebellec,parigot}@minos.inria.fr.

- Some languages, for instance Lisp, are dynamically- and/or weakly-typed, which makes many errors apparent only at run-time, if ever.
- When the AG-description language is derived from an existing implementation language, the system has no control over the semantics of the latter and may thus be prevented from performing useful optimizations. In addition, the generated attributes evaluators can only be produced in that implementation language, which hampers the portability of the applications.
- The AG formalism does not lend itself naturally to modular programming: a complete application must generally be specified as a single, monolithic block which can be quite large. Furthermore, this problem is aggravated by the huge level of detail which is necessary to completely specify an AG: *every* attribute occurrence must be defined by a semantic rules. There exist techniques that alleviate this problem by automatically generating a large proportion of the *copy rules*, but it remains that many non-copy rules "look the same" and you still have to write them many times.
- An AG is also often difficult to read because it is based on a "concrete" grammar which has to be suitable for some parsing technique. This may lead to problems such as the duplication of many productions and associated semantic rules, the splitting of closely related computations among different productions, etc.
- It is generally argued that an AG is "structured" by the underlying grammar. This however imposes a "syntactic" view of the AG, while it would sometimes be more useful or pleasant to have a "semantic" view of it.

These drawbacks are quite unfortunate because they severely undermine the attractiveness of the AG formalism, especially for large projects. Hence, as part of the development of the new FNC-2 AG-processing system [JoP89, JPJ90], aiming at production-quality, we undertook the design of a new, specialized language for specifying AGs which would address these problems. We called it OLGA.[1] Its detailed specification appears in [JoP89, part II].

Our requirements for OLGA were threefold: it should be a useful and pleasant *tool* for the AG-developer; it should be the best suited partner to the FNC-2 system, from the internal point of view; it should be a "nice" language. This translated into the following set of desirable qualities:

- The language should be *easy to learn*, i.e. it should have a "natural" semantics.
- It should also provide the AG-developer with as much *programming safety and reliability* as possible, while remaining very *versatile*. It should also provide *helpful* features.
- It should also allow the development of very *readable* and *modular* programs.
- It should be as *close* as possible *to Knuth's theory of attribute grammars*, providing all their expressive power but no more; constructs involving side-effects should therefore be banned.

[1]OLGA is the acronym of the French wording "Ouf! un Langage pour les Grammaires Attribuées" which means "At last! a language for attribute grammars."

- The language should remain *independent from the evaluation method* which will eventually be chosen (e.g. exhaustive vs. incremental). Similarly, it should remain independent from, but *easily and efficiently translatable into*, most implementation languages (the first targets we consider are C, Lisp and Ada).
- And, in the end, when all of the above requirements are satisfied, the last features of that ideal language should be *simplicity* and *cleanness*.

Of course, these requirements in many cases contradict one another. Arbitrating between them is the crux of language design [Hoa73, Ten81].

The rest of this paper will briefly present the current definition of OLGA, both as a general-purpose applicative language (section 2) and as a specialized language for AG-description (section 3), compare it with other works (section 4), sketch its implementation embodied in the FNC-2 system (section 5) and summarize our experience with both of them (section 6). Unfortunately, due to lack of space, only a few examples will be given (see [JoP89] for detailed ones).

2 OLGA as a General-purpose Applicative Language

2.1 Applicative Character

First of all, OLGA is a *purely applicative* language—well, nearly purely, see below—which means that there is no assignable variable and side effects, just pure expressions and functions. This is a strong step towards programming safety and reliability. However, OLGA is not a *functional* language: functions are *not* first-class values which could be passed as parameters, created by functions or stored as attribute values; this is because we have found no universal way—yet—to implement them efficiently in all the implementation languages we target (but see section 7).

The basic objects of OLGA are thus values and functions. There are no control-flow constructs but value-selection ones; iteration is replaced by recursion, although there is also a `map list` construct.

2.2 Type System

OLGA is a *strongly typed* language, the other necessary condition for programming safety. Basic (predefined) types are integers, reals, booleans, characters, strings, (lexical) tokens, source indexes (positions in the source text, to output error messages) and the type *void* with one value *null*. User-defined types are enumerations, subranges of scalar types, records, discriminated unions, sets of scalar values and homogeneous lists. Structured types may be recursive. Each type definition involves the definition of corresponding construction/conversion functions. In addition, in a "functional" AG there are tree types and tree values (see below).

Functions, including user-defined ones, may have a *polymorphic* profile as in ML [CHP85, HMM86], although OLGA polymorphism is restricted to set and list types. We have a *type inference* algorithm [Car87, Mil78] which can detect at compile time every inconsistency and error and makes most type declarations *optional*.

2.3 Scope Rules and Modularity

OLGA is a *block-structured* language, with various kind of blocks: those with only implicit declarations, e.g. the `let...in` construct; those with explicit local declarations and import clauses (see below), e.g. functions bodies and productions; lastly, top-level compilation units, i.e. modules, AGs and AASes (see below). Modules and AASes export the objects they define.

The basic scope rules are similar to those of Ada [ADA 83]. As in Ada also, function names can be *overloaded*; we found this feature very convenient to enhance the readability of OLGA programs. Several distinct name spaces are used in OLGA, which allows to give the same name to different objects.[2]

OLGA supports the notion of *modules*, in which one can define a set of related objects and export them to the outside world. OLGA modules are similar to those found in e.g. Ada and Modula-2. A module is split in two actual compilation units, a *declaration module* (DCM) which declares the objects which are visible from outside, together with what is necessary to use them (e.g. function signatures), and a *definition module* (DFM) in which the actual implementation of these objects (and maybe other, non-visible objects) is given. Type checking is performed across module boundaries. Types and other objects in a declaration module may be specified as *opaque*. In addition, a module may be *parameterized* by types and/or functions. This supports the notion of abstract data types.

2.4 Miscellaneous Constructs

OLGA provides a powerful *pattern matching* construct, unifying the structural matching of ML [HMM86], the selection driven by a scalar value, such as found in many imperative languages, and the test for a union tag. For instance, given the following declarations:

```
type
   RT = record x: int; y: UT; end record
   and { relates mutually recursive types }
   UT = union void; RT; end union;
value
   v: UT;
```

you could write:

```
case v is
   UT(z: void): ...;
   UT(RT([1..10, 13], UT(y: RT))): ...;
   ...
end case;
```

The second clause matches when `v`'s tag indicates that it actually is a record of type `RT`, with the value of its first field being between 1 and 10 or equal to 13 and the tag of its

[2]For instance the definition of a type automatically defines one or more construction function with the same name as the type.

second field also indicating that it is of type `RT`; the value of the latter will be bound to the variable `y` in the right-hand side of the clause.

OLGA also provides for the declaration, activation and handling of *exceptions*. User-defined exceptions may return parameters.

2.5 Deviant Constructs

In spite of our strong will, there exist a number of situations in which programming in a purely applicative fashion proves cumbersome, in particular because some notion of side effect or global state is most natural. There hence exist in OLGA a number of constructs which violate the applicative character:

- production of *error messages*,
- interface with *external functions*, written in some foreign language,
- construction of *circular structures*,

together with an explicit sequential composition operator.

As for circular structures, we initially came up with a purely applicative construct, inspired by the `letrec` construct of ML but for values. However, it seems that this construct is impossible to compile in the general case, even with strong restrictions on the expressions involved in the cycle. So we had to introduce a much more imperative physical-replacement operation, named `define`, together with a polymorphic `undef` value. Given the following declaration:

```
type
   RT = record x: int; y: RT; end record;
```

imagine that you want to construct and return a value of type `RT` whose second field is itself; the applicative and non-applicative ways to achieve that are as follows:

```
letrec v := RT(1, v) in v

let v := RT(1, undef) in (define v.y := v ;; v)
```

The `define` "statement" raises an exception when the "variable" it modifies has a value other than `undef`.

3 OLGA as an Attribute Grammar Description Language

3.1 Abstract syntax

The syntactic base of AGs written in OLGA is not a concrete syntax as in the classical framework [Knu68] but rather an *abstract syntax* as in [VM 82], that is a grammar expressing the structure of the notions of the language being described rather than their textual appearance. This frees the grammar writer from cumbersome syntactic constraints which stem from a particular parsing method and, on the other hand, allows smaller trees since they do not include keywords, simple productions and such spurious things.

The abstract syntax formalism used for OLGA is very close to Metal [KLM83], the one used in the CENTAUR programming environment [BCD88], although with a stronger notion of typing. It provides for heterogeneous, fixed arity *operators*, similar to concrete productions, and homogeneous, variable-arity operators (*list nodes*). Operators are grouped into *phyla*, which roughly correspond to non-terminals; it is possible to include a phylum into another one.

3.2 Modularity

FNC-2 views an AG as specifying a mapping from one *attributed abstract tree* (AAT) to zero, one or more other AATs. This is a very general and versatile framework, since many language processors (e.g. compilers, specialized editors, pretty-printers, cross-referencers, etc.) use some kind of trees as internal representation of their data; FNC-2 can hence be interfaced with many such tools. FNC-2 provides a cousin language of OLGA, named *asx*, to describe those AATs by means of an attributed abstract syntax (AAS) (see also section 5). To improve modularity, an AAS may be imported into another one. The types of the attributes of an AAS must be defined in a separate DCM.

The notion of AAS is central to the modularity features of OLGA and FNC-2. In particular it is possible to "pipe" two attribute evaluators generated by FNC-2, provided that the output AAS of the first one is the same as the input AAS of the second one. This allows to achieve a high degree of modularity since it is possible to split a large, complicated AG performing several tasks into a sequence of several smaller and simpler AGs, each performing only one task (this is the notion of *attribute coupled grammars* [GG 84]). Furthermore, this is achieved at no loss of efficiency since, through *descriptional composition* [GG 84, Gie 86], it is possible to mechanically construct, from two "piped" AGs, a third one performing the same translation but without building the intermediate tree. We have proved that the unusual features of OLGA for tree construction and the AG class accepted by FNC-2 do not impair this process, however we have not (yet) implemented it.

There are two kinds of AGs in OLGA, side-effect and functional, respectively roughly equivalent to classical AGs [Knu68] and ACGs [GG 84]. A side-effect AG is one in which there is exactly one output AAS which has exactly the same syntax as the input one (but possibly different attributes); in that case, the output AAT is physically the same as the input one, except that the output attributes have replaced the input ones. A functional AG has zero, one or more output AASes; the corresponding output AATs must be constructed by semantic rules and carried by attributes, and the input AAT is destroyed thereafter. Each output AAS defines a set of *tree types* corresponding to its phyla and a set of *tree construction functions* corresponding to the operators. The output AATs must be constructed bottom-up and carried by synthesized attributes, but pattern matching can be used to access the subtrees of an already constructed tree to reuse them to construct a different tree.

An AG is structured into *phases* and, of course, productions. A phase is purely a structuring construct which has no effect on the AG in the classical sense (i.e. it is "transparent" to productions and semantic rules), except that it is a block and hence may contain local declarations and import clauses. For instance, an AG describing the verification of the contextual constraints for some programming language might have a phase for name analysis and one for type analysis; each phase needs not know about the functions used in the other, and they communicate only through the attributes.

Each production is also a block; values local to a production may depend on attributes of this production and hence play the role of what is usually referred to as "local attributes." Thus, the condition that an AG be in normal form is not too constraining. A production may appear several times in each AG or phase when this improves the readability but, of course, a given attribute should be defined only once.

3.3 Attributes and Semantic Rules

In each AG there are three kinds of attributes. The *input* and *output* attributes are those carried by the input and output AATs and are declared in the corresponding AASes. Input attributes are constants. In side-effect AGs, output attributes must be given a *direction* (inherited or synthesized) and be defined by semantic rules, whereas in a functional AG they have no direction and must be attached to the corresponding AAT while it is being built. *Working* attributes are "local" to the AG and correspond to classical ones; for instance they carry the output AATs during their construction. They must have a direction.

In OLGA, a semantic rule is written, as expected, as the assignment of some value to some attribute occurrence. The right-hand side expression can refer, in addition to attributes of the production at hand, to attributes of nodes upward in the tree and attributes of subtrees (after suitable pattern matching to control the access). OLGA provides several special constructs for semantic rules in list productions [VM 82]. The basic idea is that all the sons must be treated in a uniform way, except maybe the leftmost and rightmost ones. Thus, to define an inherited attribute of a son, the `case position` construct distinguishes the `first`, `last` and `other` sons; the corresponding expressions may refer to attributes of the father and the `left` and `right` brothers (where sensible). In the same vein, to define a synthesized attribute of the father, the `case arity` construct allows to distinguish special numbers of sons (usually, but not only, zero and one) and the general case, and to refer to attributes of the `first` and `last` sons. There is also a `map` construct to compute a value by applying some function uniformly and iteratively to synthesized attributes of the sons. Fig. 1 presents a (hopefully self-explanatory) example of these constructs; notice that, with suitable declarations for `$in-env` and `$out-env`, the first two rules would be automatically generated by FNC-2.

3.4 Automatic Generation of Unspecified Semantic Rules

As stated in the introduction, an AG is generally difficult to read because there must be a semantic rule for every attribute occurrence. Even with a mechanism to automatically generate most copy rules, such as those implemented in DELTA [Lor 77], FNC/ERN [Jou 84c] and of course FNC-2, there remains a large number of non-copy rules which "look alike" in that they express the same computation, with only syntactic variations. For instance, in an AG describing a code generator, there are many rules which specify that the code for the father is a mere concatenation of the code for the sons.

```
where decls -> DECL* use
   $in-env(DECL) := case position is
            first: $in-env(decls);
            other: $out-env(DECL.left);
         end case;
   $out-env := case arity is
            0: $in-env;
            other: $out-env(DECL.last);
         end case;
   $correct := map left & value true
            other $correct(DECL)
         end map;
end where;
```

Figure 1: Examples of semantic rules for a list production

Thus, a way to decrease the size of an AG and improve its readability is to specify in a single place a "template" for such similar rules and have the system generate all the corresponding actual rules by interpreting the template in the context of specific productions. This idea is not original [DuC88, Tie87] but is nevertheless quite recent. We implemented it in OLGA [Le B89]. The basic mechanism is the definition of *attribute classes*, which are sets of attribute occurrences, and associated templates to specify the semantic rules which define these occurrences. The templates are actually structured in two levels, a syntactic level used to specify in which productions the templates will be applied, and a semantic level which correspond to the actual rules. We needed this separation, also found in [DuC88] but not in [Tie87], because of list operators [Le B89].

The other benefit of this feature—the most important in our opinion—is that this gives a more "semantic" view of the AG, because closely related semantic computations are grouped in a single place (the definition of a class) rather than being spread over several syntactic productions. The phase construct supplements this semantic structuring.

4 Comparison with Other Languages

4.1 General-purpose Applicative Languages

The general-purpose features of OLGA borrow much from Lisp and above all ML. The similarities with the latter are so striking that we were often asked, "Why don't you use ML as the base language for OLGA?" The answer is that we did not want to be tied to a particular implementation language, even such a good one as ML and even if this often means reinventing the wheel.

However OLGA is *not* ML. First, it is not interactive. Secondly, OLGA is not functional; there is a `map list` construct to apply a function to each element in a list but you cannot define it in OLGA. There are also differences in the type system and in the notion of modules (the OLGA notion of parameterized modules is clearly closer to that of Ada and inferior to the ML notion of signatures). Generally speaking, we admit that Olga is inferior to ML regarding its formal semantic foundations. However, experience shows that it is nevertheless quite usable and that the features we did not include in Olga make it easy to implement. In addition, the syntax and concepts of Olga make it rather close to (a subset of) Ada, which should ease the teaching/learning process, in particular in industrial contexts.

4.2 AG-description Languages

As can be seen in [DJL88], most AG-processing systems use as input language a variation of the language in which the generated evaluators are written. We have shown in the introduction that this lead to serious drawbacks. In this section we compare Olga with three AG-description languages which also try to avoid this pitfall: ALADIN [KHZ 82], the language of the GAG system, the anonymous language of the Linguist system [Far 89], hereafter referred to as Linguist, and SSL [RT 89b], the language of the Synthesizer Generator [RT 89a].

All four languages are basically applicative, although all of them provide some deviant constructs such as an interface with external functions. All of them provide for recursive function definitions and offer more or less sophisticated selection constructs (the richer being Olga). The type constructors of ALADIN and Linguist are roughly the same as those of Olga, except that Linguist also provides functional types and values. Only Olga provides polymorphism. The SSL type system is rather different since it integrates both the syntactic (trees) and semantic (attributes) domains, i.e. the value of an attribute can be some subtree, itself attributed, on which you can evaluate an AG; hence all type constructors are presented in terms of phyla and operators (SSL also uses an abstract syntax, although with no list construct). This idea is pushed to its limits in MARVIN [GGV 86], which provides only one type constructor, grammars, and one way to define functions, AGs (more precisely ACGs).

As for productions, attributes and semantic rules, only Linguist remains absolutely faithful to the original framework, by providing pure BNF; this is related to the fact that it is the only system which actually integrates a parser generator. SSL also provides pure BNF, but its grammars are used as abstract ones, i.e. they do not necessarily reflect the textual appearance of the source language (the parsing aspect is rather separate from the main body of an SSL AG). ALADIN provides extended BNF, i.e. the RHS of a production may be a regular expression; however alternatives are forbidden (they must be separate productions) and repetition clauses may not be nested, so the expressive power is roughly similar to that of Olga list operators. Special laws deal with the attributes of symbols appearing in a repetition clause, some of them involving implicit list processing, but, in our opinion, these laws are not very natural and hard to understand and make the AG difficult to read. All languages but Linguist provide access to upward non-local attributes; ALADIN also allows to access downward non-local attributes, but we think that this construct is rather ill-defined. All four languages offer a construct to abbreviate or a mechanism to automatically generate copy rules, but it seems that Olga is the winner in this respect. Linguist also provides a very crude mechanism to generate some non-copy rules. ALADIN is the only language to provide *context conditions*, i.e. predicates on attributes values which must evaluate to true for the semantic rules to be valid; we believe however that this does more harm than good, because the AG author has no control on error recovery.

There exist several "external" means by which an AG system can provide more or less modularity:

- define external functions in separate files;
- file inclusion à la *cpp*.

ALADIN, Linguist and SSL all provide for the former, but these external functions *must* be written in a foreign language; hence you are forced to trade modularity for the comfort of using the built-in language. Linguist is designed to be used with *cpp* whereas SSL has its own file inclusion mechanism. An interesting feature of Linguist is that it allows to encapsulate the translation defined by a whole AG into a single function, which can then be used as an external function by another AG [FaS89]; this allows some form of modularity à la MARVIN but is not well integrated with the language itself. It hence appears that OLGA is clearly superior to its three competitors on modularity; only MARVIN can compete with OLGA in this respect, thanks to ACGs composition, but it is otherwise only a minimal language.

5 Implementation: the FNC-2 System

The FNC-2 system [JoP89, JPJ90] is the implementation of OLGA, that is an AG-processing system. It aims at production-quality by being powerful, efficient, easy to use and versatile. More details on the evaluation methods and optimization techniques used in FNC-2 to achieve these qualities appear in [JPJ90, JuP90, Par 88]. Its easiness of use is due to the qualities of OLGA, and its versatility stems from the view that an AG is an AAT to AAT mapping, which allows to integrate evaluators generated by FNC-2 with many other tree-based tools, and from the complete independence of an AG specification in OLGA w.r.t. its implementation(s).

FNC-2 comes with several companion processors that allow the development of complete applications. The organization of the various components forming a typical application is shown in Fig. 2. OLGA is used to specify the most important of these components, AGs and modules; the same compiler is used for both—the evaluator generator is merely bypassed when compiling a module. As said above, *asx* describes AASes. *Atc* describes tree construction "actions" associated to the productions of a concrete grammar; there exist two instantiations of this language, one on top of the SYNTAX[3] system [BoD88] and one of top of *yacc*. *Ppat* [Pla89] describes the textual layout (unparsing) of AATs, using the TEX-like paradigm of nested boxes of text. *Mkfnc2* describes the organization of the various components of the application; in the future, the corresponding processor will be extended into a full-fledged library manager. All of these specialized languages have much in common with OLGA, which makes the whole system easy to use and smoothly integrated.

As of today, FNC-2 is in a quite usable state, even for large applications (see below); an evidence of this is that all the components of the system, except the evaluator generator, have been or are being bootstrapped. However, not all of OLGA is implemented

[3]Trademark of INRIA

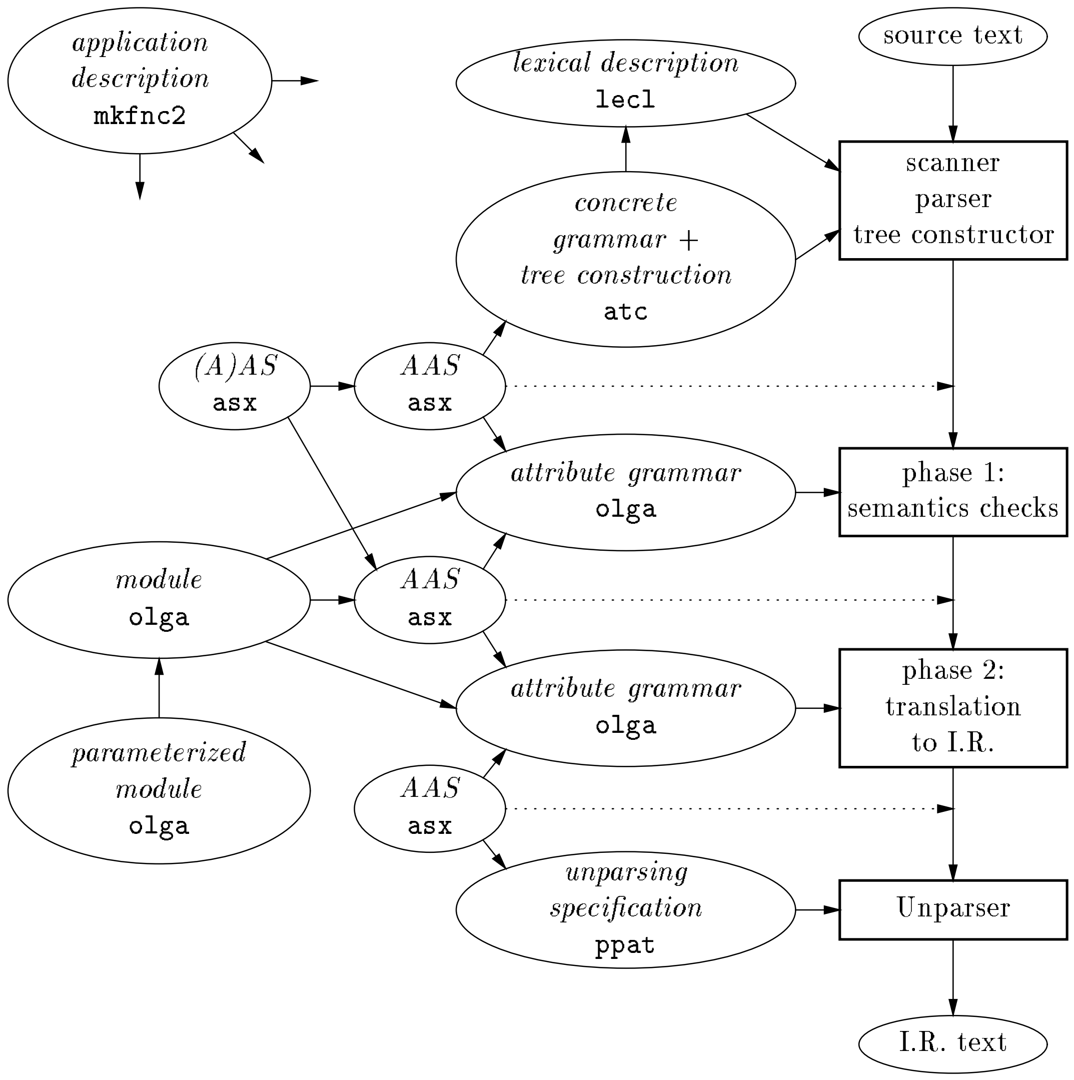
application description mkfnc2
lexical description lecl
source text
scanner parser tree constructor
concrete grammar + tree construction atc
(A)AS asx
AAS asx
attribute grammar olga
phase 1: semantics checks
module olga
AAS asx
attribute grammar olga
phase 2: translation to I.R.
parameterized module olga
AAS asx
unparsing specification ppat
Unparser
I.R. text

Figure 2: A Typical Application

yet: full polymorphism, parameterized modules, attribute classes and the associated rule templates, and some other features are still missing. A rather naive translator to Lisp has been implemented. The translator to C is more sophisticated but still lacks a garbage collector. It performs classical optimizations such as tail-recursion elimination. *Ppat* is still under development.

6 Experience

6.1 Design and Implementation

We tried to design Olga without thinking of a particular implementation; we believe that the Ada experience was an example to follow in this respect. Indeed, the first versions of the Olga manual were around long before FNC-2 began to run. This had two advantages: first, this leads to a cleaner definition of the language, because you are not hampered by implementation constraints; second, provided that the definition remains stable enough, it allows to start to use the language early and hence have real-size examples to test the system when it comes up (this was the case for FNC-2 with the PARLOG compiler described below). However, this may lead to overlook implementation problems which may in the end prove impossible to overcome (e.g. the `letrec` construct of OLGA). Maybe we would not have had such difficulties if we had chosen to base our language on a "good" existing language such as ML; however we believe that our approach leads to more advantages than drawbacks, as we tried to show in this paper.

It seems that we fell in all the usual traps of language design [Hoa73, Ten81]. In particular, we have never been able to simplify the language (except in one case, the pattern matching construct alluded to above), and we always ended up in adding new constructs which proved very hard to integrate smoothly with the rest of the language. This is made especially acute by our high ambitions for OLGA...

Our philosophy for system development was to start with a minimal, quick and dirty implementation and then use bootstrap to pursue the development with better conditions (after all, the main applications of AGs are language processors such as... AG processors!). However, non-trivial optimizations such as tail-recursion elimination and space optimizations in the evaluator generator proved to be necessary right from the beginning.

OLGA proved easy to translate to Lisp, because of their common concepts and features and because there is no need for a particular run-time system. Naively translating it to C was also easy, but making it efficient is hard; in particular, we need a garbage collector, and writing such a tool in a high-level language is not easy.

6.2 Language Usage

Until now we have worked on three real-size applications:

- a compiler from the parallel logic language PARLOG to code for the SPM abstract machine [GJR 87];
- a compiler from ISO-Pascal to P_code, which had been originally written in Lisp for the FNC/ERN system [Jou 84c] and was translated to OLGA [Dur88] (work not yet completed);
- and FNC-2 itself (bootstrap) and its companions (*asx*, *atc*, *ppat*, *mkfnc2*).

In addition, FNC-2 is used in the PAGODE code generator generator [DMR89], both to develop the generator itself and to compile some of the modules it generates.

Even in its incomplete present status, we found OLGA generally satisfying. In particular, there are very few parts of the system which could not be written in OLGA; the most prominent are the evaluator generator and the parts which write text to files (*ppat* will provide a solution for those). This shows that the expressive power of the language is sufficient for our purposes. The automatic generation of most copy rules and many non-copy rules greatly improves the readability of the AGs, especially for code-generation-like applications. But the single most beneficial feature of OLGA is its constructs for modularity: presently there are about 30,000 lines of OLGA, *asx* and *atc* code in the system, summarized in Table 1 (these exclude the *ppat* subsystem, the files processed by SYNTAX, and

	# files	# lines			
		min.	max.	total	ave.
AGs	7	354	3,212	10,118	1,445
AASes	8	8	381	779	97
decl. mod.	15	28	391	2,891	193
def. mod.	15	55	3,188	13,404	894
atc	4	60	2,089	2,575	644
total	49	8	3,212	29,767	607

Table 1: Source files in the FNC-2 system

the C files). As can be seen, the files are rather small and hence very readable. Even the 3,212-lines AG (the OLGA static semantics checker) is easy to understand because it is split in two phases (name analysis and type checking) which address only one problem. We would have had much greater problems if we had had to specify the FNC-2 system in a single AG. Furthermore, separate compilation comes with (true) modularity, and this saves much time in the development process.

We pushed the usage of AGs and modularity very far. For instance, the *ppat* subsystem is composed of 9 files, including one bootstrapped file and one reused from FNC-2 itself, and the pretty-printers it generates are composed of one actually generated file and 5 files which are reused in every pretty-printer. Each one of them is hence rather small and easy to read.

It is quite hard to measure the efficacy of a programming language, i.e. the relative productivity of a programmer using this language. Let us however quote a single figure regarding OLGA: the above-described 30,000 lines, together with other support files, were written and tested by one man in not much more than one year.

7 Conclusion and Future Work

We presented in this paper the design, implementation and preliminary evaluation of the new OLGA AG-description language. Even though the language may be improved and FNC-2 is still under development, we are generally satisfied with both of them.

There are a number of constructs that qualify for possible inclusion into OLGA, most notably: unification of the syntactic and semantic domains, as in SSL; functional types and values, as in Linguist[4] and ML; multi-valued expressions, functions and semantic rules, also as in Linguist.

Much remains to be done on the implementation of FNC-2. We have to implement all of OLGA. We also have to improve the efficiency of the C implementation; the run-time system of "Standard ML of New Jersey" [App89] is a rich mine of very good ideas in this respect. Generally speaking, we have to study and implement well-known techniques for efficient implementation of applicative languages on conventional architectures. It is also planned to integrate FNC-2 with CENTAUR and the GIGAS system [Fra89] to obtain a system with aims similar to those of the Synthesizer Generator but with greater

[4]The Linguist implementation of functional values seems fairly simple, so our arguments against them may prove irrelevant.

performances and facilities.

When all of this is completed, FNC-2 will really be one of the very few production-quality AG-processing systems in the world.

References

[ADA 83] American National Standards Institution, *ADA Reference Manual* ANSI/MIL-STD 1815A, Jan. 1983.

[App89] A. W. Appel, "A Runtime System," Princeton Univ., draft, Feb. 1989.

[BCD88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual, "CENTAUR: the System," *SIGSOFT Software Eng. Notes* **13** (Nov. 1988), 14–24.

[BoD88] P. Boullier & P. Deschamp, *Le système SYNTAX—Manuel d'utilisation et de mise en œuvre sous Unix*, INRIA, Rocquencourt, Sept. 1988.

[Car87] L. Cardelli, "Basic Polymorphic Typechecking," *Sci. Comput. Programming* **8** (Apr. 1987), 147–172.

[CHP85] G. Cousineau, G. Huet & L. Paulson, *The ML Handbook*, INRIA, Rocquencourt, May 1985, Co-published by Univ. of Cambridge.

[DJL88] P. Deransart, M. Jourdan & B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. **#323**, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[DMR89] A. Despland, M. Mazaud & R. Rakotozafy, "Using Rewriting Techniques to Produce Code Generators and Proving them Correct," INRIA, Rapport RR-1046, Rocquencourt, June 1989, To appear in *Sci. Comput. Programming*.

[DuC88] G. D. P. Dueck & G. V. Cormack, "Modular Attribute Grammars," Univ. of Waterloo, research report CS-88-19, May 1988.

[Dur88] O. Durin, "Traduction en OLGA d'une grammaire attribuée écrite en Lisp," École Polytechnique, rapport de stage d'option, Palaiseau, July 1988.

[Far 89] R. Farrow, *The Linguist Translator-writing System—User's Manual* version 6.25, Declarative Systems Inc., Palo Alto, CA, June 1989.

[FaS89] R. Farrow & A. G. Stanculescu, "A VHDL Compiler based on Attribute Grammar Methodology," *ACM SIGPLAN Notices* **24** (July 1989), 120–130.

[Fra89] P. Franchi-Zannettacci, "Attribute Specifications for Graphical Interface Generation," in *Information Processing '89*, G. X. Ritter, ed., North-Holland, Amsterdam, Aug. 1989, 149–155.

[GG 84] H. Ganzinger & R. Giegerich, "Attribute Coupled Grammars," *ACM SIGPLAN Notices* **19** (June 1984), 157–170.

[GGV 86] H. Ganzinger, R. Giegerich & M. Vach, "MARVIN: a Tool for Applicative and Modular Compiler Specifications," Fachbereich Informatik, Univ. Dortmund, Forschungsbericht 220, July 1986.

[GJR 87] J. Garcia, M. Jourdan & A. Rizk, "An Implementation of PARLOG Using High-Level Tools," in *ESPRIT '87: Achievements and Impact*, Commission of the European Communities—DG XIII, ed., North-Holland, Amsterdam, Sept. 1987, 1265–1275.

[Gie 86] R. Giegerich, "On the Relation between Descriptional Composition and Evaluation of Attribute Coupled Grammars," Fachbereich Informatik, Univ. Dortmund, Forschungsbericht 221, July 1986.

[HMM86] R. W. Harper, D. B. MacQueen & R. Milner, "Standard ML," Lab. for Foundations of Comp. Sc., Dept. of Comp. Sc., Univ. of Edinburgh, report ECS-LFCS-86-2, Mar. 1986.

[Hoa73] C. A. R. Hoare, "Hints on Programming Language Design," Comp. Sc. Dept., Stanford Univ., tech. report STAN-CS-73-403, Oct. 1973.

[Jou 84c] M. Jourdan, "Les grammaires attribuées: implantation, applications, optimisations," Univ. Paris VII, thèse de Docteur-Ingénieur, May 1984.

[JoP89] M. Jourdan & D. Parigot, *The FNC-2 System User's Guide and Reference Manual*, INRIA, Rocquencourt, Feb. 1989, This manual is periodically updated.

[JPJ90] M. Jourdan, D. Parigot, C. Julié, O. Durin & C. Le Bellec, "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System," *ACM SIGPLAN Notices* **25** (June 1990), 209–222.

[JuP90] C. Julié & D. Parigot, "Space Optimization in the FNC-2 Attribute Grammar System," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990.

[KLM83] G. Kahn, B. Lang, B. Mélèse & É. Marcos, "Metal: a Formalism to Specify Formalisms," *Sci. Comput. Programming* **3** (1983), 151–188.

[KHZ 82] U. Kastens, B. Hutt & E. Zimmermann, *GAG: A Practical Compiler Generator*, Lect. Notes in Comp. Sci. #**141**, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[Knu68] D. E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2** (June 1968), 127–145, Correction: *Math. Systems Theory* **5**, 1 (Mar. 1971), 95–96.

[Le B89] C. Le Bellec, "Spécification de règles sémantiques manquantes," Dépt. d'Informatique, Univ. d'Orléans, rapport de DEA, Sept. 1989.

[LMW88] P. Lipps, U. Möncke & R. Wilhelm, "OPTRAN – A Language/System for the Specification of Program Transformations: System Overview and Experiences," in *Compiler Compilers and High Speed Compilation*, Dieter Hammer, ed., Lect. Notes in Comp. Sci. #**371**, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1988, 52–65.

[Lor 77] B. Lorho, "Semantic Attributes Processing in the System DELTA," in *Methods of Algorithmic Language Implementation*, A. Ershov & Cornelius H. A. Koster., eds., Lect. Notes in Comp. Sci. #**47**, Springer-Verlag, New York–Heidelberg–Berlin, 1977, 21–40.

[Mil78] R. Milner, "A Theory of Type Polymorphism in Programming," *J. Comput. System Sci.* **17** (Dec. 1978), 348–375.

[Par 88] D. Parigot, "Transformations, évaluation incrémentale et optimisations des grammaires attribuées: le système FNC-2," Univ. de Paris-Sud, thèse, Orsay, May 1988.

[Pla89] É. Planes, "PPAT: un décompilateur d'arbres attribués pour le système FNC-2," Dépt. d'Informatique, Univ. d'Orléans, rapport de DEA, Sept. 1989.

[RT 89a] T. Reps & T. Teitelbaum, *The Synthesizer Generator*, Springer-Verlag, New York–Heidelberg–Berlin, 1989.

[RT 89b] ———, *The Synthesizer Generator Reference Manual* 3rd edition, Springer-Verlag, New York–Heidelberg–Berlin, 1989.

[Ten81] R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Tie87] M. D. Tiemann, "Removing Redundancy in Attribute Grammars," Parallel Processing Program, Microelectronic and Computer Technology Corp., manuscript, Austin, TX, July 1987.

[VM 82] A. O. Vooglaid & M. B. Mériste, "Abstract Attribute Grammars," *Progr. and Computer Software* **8** (Sept. 1982), 242–251.