The LDL - Language Development Laboratory

Günter Riedewald

Universität Rostock, Fachbereich Informatik
A.-Einstein-Str. 21, O-2500 Rostock, Germany
E-mail: riedewald@informatik.uni-rostock.dbp.de

Abstract. LDL is a system supporting the design of procedural programming languages and generating interpreters for prototyping purposes. Moreover, test sets for testing interpreters/compilers of the developed language can be generated. LDL is based on GSFs - a kind of attribute grammars - and uses the denotational approach for semantics definition. For the prototype interpreter its correctness can be proved.

1 Introduction

It is an old dream of compiler designers that formal specifications can at the same time serve as programming language definitions and as compiler specifications. The goal is to generate automatically realistic compilers from formal language specifications. A good survey about some older attempts based on either language definitions with denotational semantics or algebraic language definitions is represented in [J 80]. E.g. the systems of Jones/ Schmidt and Mosses are based on denotational semantics, whereas Morris and Gaudel prefer the algebraic approach. Newer papers or projects are described in e.g. [A 86], [W 86], [BP 89], [BA 90], [FL 90]. Although the final goal is to generate realistic compilers at the moment it is rather realistic to generate prototype compilers. But it seems that Lee's project MESS ([L 89]) is very near to the goal.

Our system LDL exploits the concept of [R 91], i.e. a language is defined by a GSF, a kind of attribute grammars, and applies the denotational approach for semantics definition. Because GSFs are closely related to PROLOG programs, after some modifiations and extensions the language definition is turned into a PROLOG program, which can be considered as a prototype interpreter of the defined language. The correctness of the prototype interpreter can be proved in an analogous way as in [ADJ 80].

The language definition serves also as an input of a test-case generator. The generated test programs satisfying context conditions can be used to test compilers/interpreters constructed for the developed language applying any compiler compiler. For this purpose the test programs are executed once by the prototype interpreter and once by the tested compiler/interpreter. Because the prototype interpreter is correct, from comparing the results of both executions it can be seen whether the tested compiler/interpreter works correctly for the test programs. Additionally, more sophisticated programs suggested by language users can be used. In such a way LDL supports the development of correct compilers/interpreters to a high degree through validation, verification and testing.

Keeping in mind Koskimies' statement ([K 91]) "The concept of an attribute grammar is too primitive to be nothing but a basic framework, the 'machine language' of language implementation." LDL offers a higher-level tool supporting the definition of procedural languages and their implementation in form of a prototype interpreter. For this purpose the LDL library contains predefined language constructs together with their denotational meaning expressed by PROLOG clauses. At present this library contains all usual PASCAL constructs excluding structured data consisting of compound components.

The structure of the LDL system is described in section 2. In section 3 some information about future work will be given. GSFs and the relations to other formalisms are described e.g. in [R 91].

2 Structure of LDL

The main components of LDL are the following:
- a *tool for language design* which is based on a *library of language components* and a *knowledge base*
- a *test set generator* for the generation of program examples which satisfy all context conditions of the defined language.

The *tool for language design* supports the definition of procedural languages and the design of prototype interpreters for the defined languages. Within LDL GSFs are used for language definition, where the semantics is defined in a denotational-like style. The GSF can be combined from predefined rule sets, but it is also possible to define new rules. The syntax of the language is described by the context-free basic grammar of the GSF. The context conditions are realized by auxiliary syntactical functions and the semantics of the language is defined by semantic functions in the sense of GSF. If the language designer uses predefined GSF rules then also context conditions and semantics are included. Defining new GSF rules, either predefined or new auxiliary syntactical and semantic functions can be applied.

The *knowledge base* contains some knowledge about procedural programming languages in general and about the predefined components (i.e. GSF rules and functions) for building prototype interpreters/compilers in particular. That knowledge is used by the tool for language design in order to direct the language designer and to guarantee (or to support at least) a consistent language definition in that sense, that the application of contradictory concepts (e.g. static and dynamic binding of global variables in procedures) is prevented.

Since semantics definitions within LDL are based on the denotational approach, the system offers predefined functions representing fundamental concepts of denotational semantics. Moreover, there are help functions providing more qualified services which are usually necessary in order to design nontrivial procedural languages, e.g. functions representing a block concept. All these functions are defined by PROLOG clauses and included in the *library of language components* .

A first result of using the tool is a GSF scheme in form of a PROLOG program describing both the syntactical and semantic structure of the developed language. This GSF scheme is sufficient to compute the meaning of each program in form of a term consisting of semantic function symbols (Using the terminology of [L 89] a GSF scheme defines the macrosemantics. The meaning of a program in form of a term is then comparable with a POT in the sense of [L 89].).

Example 1: statement(Sm) :- @ repeat, sm_list(S1), @ until, expression(_,Type,Exp), # equal(bool,Type), & repuntil(Sm,Exp,S1).

This PROLOG clause defines a repeat -statement. #equal(...) is an auxiliary syntactical function (in the GSF sense) which checks whether the type Type of the expression following the terminal until is boolean . repuntil(...) is a semantic function computing the meaning Sm of the repeat -statement depending on the meaning Exp of the expression following until and on the meaning S1 of the repeated statement sequence.

#

To get a complete *prototype interpreter* a GSF scheme must be extended by definitions of the auxiliary syntactical functions (i.e. context conditions) and the semantic functions (i.e. dynamic semantics). (It is comparable with the definition of the micro-semantics in [L 89].) Supposing the GSF scheme was assembled using the tool for language design, most of the functions appearing in the GSF scheme will be offered by the library of language components. However, there is no need to define these functions at the beginning of language design ([R 91]). E.g., the language designer could start to write down the GSF scheme in order to define the (concrete) syntax and the semantic structure of the language. Then, first source examples could be parsed without any implementation for static and dynamic semantics. In a second step the language designer could define the context conditions by implementing the auxiliary syntactical functions of the GSF. Finally, in order to get a complete prototype interpreter which allows to interprete programs of the defined language, the designer must implement the semantic functions using components from the library.

Example 2:
- The PROLOG clause equal(X,Y) :- X==Y. defines the auxiliary syntactical function equal(...).
- Supposing the meaning of a repeat -statement is expressed by the term repuntil(E,Sm), where E is the meaning of the condition of the repeat -statement and Sm is the meaning of the repeated statement sequence then the following clause desribes the interpretation of that term:

```
com(repuntil(E,Sm)) :- !, com(Sm), reval(Val,E),
          (call(Val);com(repuntil(E,Sm))),!.
```

#

The prototype interpreter operates as follows :
- A source program is read token by token from a text file.
- Each token is classified by a standard scanner. It is a part of LDL and should be sufficient for most lexical requirements. If necessary it is possible to extend the scanner (which is programmed in PROLOG, too) for other lexical classes.
- The parsing and checking of context conditions is interconnected with input and scanning. If the context-free basic grammar of the source language is an LL(k)-grammar the PROLOG system itself can be used straightforwardly for parsing, whereas LR(k) grammars demand to include a special parser into the prototype interpreter.
- Recognizing a language construct its meaning in form of a term is constructed by connecting the meaning of its subconstructs.

- The term representing the meaning of the whole program is interpreted, i.e. the function symbols of the term are considered as functions transforming a given program state into a new one, where a state is, roughly speaking, an assignation of values to program variables.

For a prototype interpreter developed using LDL, a correctness proof can be given. For that purpose, the equivalence between the formal language definition in denotational-like style and its LDL representation in PROLOG must be proved. The proof can be done in an analogous way as in [ADJ 80].

It is nearly impossible to prove the correctness of compilers, generated by compiler compilers aiming at production-quality. To test such compilers and the prototype interpreters, which were designed using LDL, the system offers the so-called *test set generator*. The aim of this additional component is to generate programs of the defined language which are syntactically correct and which satisfy the context conditions of the language. To limit the multitude of generated programs it should be possible to define some additional properties of programs, e.g. the maximum number of statements in a statement sequence, the maximum depth of nested statements or expressions. Thus, our test set generator consists of the following components:
- a GSF scheme in form of PROLOG clauses
- PROLOG clauses defining auxiliary syntactical functions and thereby context conditions
- PROLOG clauses generating source programs
- a control mechanism which guarantees that the generated programs possess the additional properties.
The needed GSF scheme will be usually the same as required for the prototype interpreter. Also the definition of the auxiliary syntactical functions can be taken straightforwardly from the language definition for the prototype interpreter. The clauses generating source programs are offered by the LDL library.

The test set generator operates as follows:
- The start symbol of the context-free basic grammar is applied in order to generate a program of the language.
- The first generation step is the generation of a term which can be considered as the meaning of a syntactically correct program satisfying the context conditions. Then, the program is derived from this term.
- The control mechanism which guarantees the additional properties of the programs to be generated is applied in interconnection with generation and testing context conditions. At present

some elements to control the generation of test programs are included into the PROLOG clauses by hand. But there are other possibilities as e.g. described in [D 91] or [Aug 91]), which could also be used to describe the desired properties of generated test programs and to control the generation.

3 State of Implementation and Future Work

The project LDL was started in 1991. First, a prototype interpreter for a toy language SPL was implemented based on a language definition using the GSF formalism and its relation to PROLOG ([R 91]). Then, this language definition was modified in such a way that it was possible to generate test programs satisfying the context conditions of SPL. To control the generation some control elements, e.g. counters controlling the number of statements in a statement sequence or the depth of nested statements or expressions, were included by hand. Based on these experiments the library of language constructs have been extended. Now, it is possible to use nearly all language constructs of PASCAL excluding structured data with structured components. Applying these language constructs a PASCAL-like language YAL was defined and tested with some programs, e.g. Ackermann's function, Towers of Hanoi, BUBBLESORT.
The implementation language is Quintus PROLOG under SUNOS, but we gained also experiences with other PROLOG systems. Using Quintus PROLOG the run-time efficiency of the prototype interpreters for the designed languages is surprisingly high.

Future work will concentrate upon the following problems:
- The tool for language design will be realized in form of an expert system.
- The test set generator must be extended for more powerful languages and also by a control mechanism not disturbing the process of language definition.
- Methods of possibly automatic test comparison must be developed.

Acknowledgement

I am very grateful to Ralf Lämmel who developed the library of language constructs and who designed and tested the language YAL.

References

[A 86] Arbab,B.: Compiling circular attribute grammars into PROLOG IBM Journal of Research and Development, 30(3),1986,294-309

[ADJ 80] Thatcher,J.W.,Wagner,E.G.,Wright,J.B.: More on advice on structuring compilers and proving them correct. In: [J 80],165-188

[AM 91] Alblas,H.,Melichar,B. (eds.): Attribute Grammars, Applications and Systems. Proceedings of the Inter. Summer School SAGA, Prague,Czechoslowakia,June 1991,LNCS #545,Springer-Verlag

[Aug 91] Auguston,M.: FORMAN - program formal annotation language In: Proc. of the 5th Israel Conf. on Computer Systems and Software Engineering,Gerclia,May 1991,IEEE Edition,149-154

[BA 90] Bretz,M.,Abels,T.: Generierung von Programmauswertern aus denotationellen Semantikbeschreibungen. In: J. Ebert (Hrsg.), Alternative Konzepte für Sprachen und Rechner,Bad Honnef 1990, Bericht 8/90,53-64

[BP 89] Bryant,B.R.,Pan,A.: Rapid prototyping of programming language semantics using PROLOG. IEEE Software,1989,439-44

[D 91] Denney,R.: Test-case generation from Prolog-based specifications IEEE Software,March 19991,49-57

[FL 90] Forbrig,P.,Lämmel,U.: Prototyping in compiler construction In: Překladače programovacích jazyků, Sborník přednášek,Praha, 1990,ČSVTS-FEL-ČVUT,174-190

[J 80] Jones,N.D. (ed): Semantics-directed compiler generation LNCS #94,Springer-Verlag 1980

[K 91] Koskimies,K.: Object-orientation in attribute grammars In: [AM 91],297-329

[L 89] Lee,P.: Realistic compiler generation. The MIT Press 1989

[R 91] Riedewald,G.: Prototyping by using an attribute grammar as a logic program. In: [AM 91],401-437

[W 86] Watt,D.A.: Executable semantic descriptions Software-Practice and Experience,16(1),13-43