PERLUETTE : A Compilers Producing System

using Abstract Data Types

Ph. Deschamp, INRIA
Domaine de Voluceau
BP 105 78153 LE CHESNAY

ABSTRACT

Real compilers are usually ad hoc programs. They are costly to write and maintain, and too much complex to be proved correct. This paper presents the compilers producing system Perluette. This system is based upon a formal semantics of programming languages. Programming languages are considered to be algebraic data types. Then it becomes possible to specify and prove their implementations as representations of an algebraic data type into another one.

This formal approach has many advantages; among these are: the modularity of the compilers specifications; the independance between the source language and the target language descriptions.

This paper gives an example of compiler specification and describes the implementation of the system in its current state.

INTRODUCTION

Compilers are extensively used in the computer scientist's daily life and all users rely heavily on them. Moreover new programming languages as well as new computers spring up every year. It is thus fundamental to reduce the cost of compiler-producing, while enabling the compiler-writers to prove that the result of their work is correct.

Our approach to compiler specification is based on algebraic abstract data types. We present here the first implementation of our compiler generator, Perluette.

In the first part of this paper we introduce the overall structure of Perluette. The second part shows how we specify compilers, by way of a small example. The last part of the paper describes the implementation we have chosen for the compilers produced by the current version of the system.

I. - COMPILER PRODUCTION : Structure of Perluette

Many techniques are known which try to automate compiler-writing [Gri 71, AU 77]. In particular, with respect to the source language:

- lexical and syntactic analyses can be performed by table-driven programs;
the analysis tables are usually produced from a grammar in BNF or from regular expressions;

- the contextual constraints on source programs can be checked using two-level grammars, affix grammars or attributes [Knu 68];

- the ways to specify how to associate a semantic value with source programs are less agreed-upon, since they are not really well-suited to compilation - we can mention axiomatic semantics [Hoa 69], denotational semantics [Ten 76], algebraic semantics [CN 76], ...

As for the target code, much research is being devoted to the problem of producing "good" (efficient, reliable) code as easily as possible. This has led to a variety of techniques; they usually use an internal form of the source program to be translated (trees or tuples).

The most conventional way to write compilers uses none of the techniques above: each compiler is an ad hoc program; source language dependencies and target language dependencies are scattered through the body of the compiler. Such compilers are difficult to modify or maintain; moreover they are not readily adapted to accept another source language, or to produce another kind of target code.

Thus it is necessary, in order to achieve the goals of cost-reduction and correctness-proof, to produce compilers from independent descriptions of the source and target languages specifications, as suggested by F.L. Morris [Mor 73]. This leads to the following "ideal" schema, where all the dirty work is automated:

![Diagram](image)

Source Language specifications ➔ Compilers ➔ Compiler
Target Language specifications ➔ Producer
Implementation Choices

The implementation choices specification describe the kind of implementation the compiler designer chooses for each object of the source language - e.g. one needs to specify if arrays are implemented row-wise or column-wise. In our opinion this is an essential part in the design of compilers, and these implementation choices must be rigorously specified and proven correct.

Our compilers Producer System, named Perluette, works basically along these lines: the source language and target language are specified by algebraic abstract data types [GH 78, GHM 78] - abridged to ADT in the sequel -; the implementation choices are specified as a representation of the source ADT by the target ADT, allowing to make use of the results of research on representations of ADTs [OTW 78]. The intermediate