FILOMAT (Niš) 9:2 (1995), 295–302

Filomat '94, Niš, October 22-24, 1994
Geometry. Computer Sciences

ON TRANSLATING MODULA-2 PROGRAMS TO C: LOCAL PROCEDURES AND MODULES

Lehel Szarapka and Dragan Mašulović

ABSTRACT. *This paper demonstrates techniques that enable efficient translation of Modula-2 programs to C. It focuses on a key problem that appears during translation: local procedures and modules. The techniques are presented via examples. For the sake of readability, instead of C a subset of Modula-2 (called Flat Modula-2) is used as a target language.*

Introduction

Modula-2 [1] is a high-level programming language designed by Prof. Niklaus Wirth at the ETH, Zürich. Its key design goal was simple and elegant support of modular programming which is the most important step towards programming in the large.

C [2] is a low-level programming language designed to help reimplementing UNIX[1]. In spite of its consistent inconsistency and poor design, C is a wide spread programming language. Because of that, it has been recognized lately as a *platform independent assembly language* thus giving rise to a slightly different approach to compilation: translation to C as a target language. Such compilers are more portable than "classic" compilers, even those which choose a form of pseudo code as a target language.

Following the tradition of Algol-like languages, Modula-2 admits declaration of procedures and modules local to other procedures. On the other hand, C does not allow declarations of functions local to other functions. Thus local modules and procedures present a key problem that a translator to C has to take care of [3, 4]. This paper presents a set of techniques that solve the problem.

1991 *Mathematics Subject Classification.* 68N20.

Key words and phrases. translation, nested procedures, nested modules.

[1]UNIX is a trade mark of AT&T Bell Labs

For the sake of readability, instead of C a subset of Modula-2 (called *Flat Modula-2*) is used as a target language. Flat Modula-2 does not allow declarations of modules, procedures, types and constants local to other procedures. Thus, translation of Flat Modula-2 programs to C is an one-one mapping and shall not be discussed here because there are several public domain translators from restrictions of Modula-2 (similar to Flat Modula-2) to C (e.g. [4]). Some examples of translating Flat Modula-2 to C can be found in Appendix A.

The rest of the paper is organized as follows: Section 1 describes two major techniques upon which the translation process is based. Section 2 handles constant and type declarations, Section 3 handles local procedures, while Section 4 discusses local modules. Section 5 concludes the paper.

1. Two major techniques

The translation process is based on two major techniques:

(1) globalization of local entities that are not local variables and

(2) systematic renaming.

The basic idea is simple: all the local entities (except local variables) are taken out of the procedure declaration and are declared globally. This is the only way to take care of local procedures and modules. At the same time, globalization gives an elegant solution to local type and constant declarations. In order to prevent name clashes, a systematic renaming is performed. Because names are of no relevance to the compiler, a brute force approach can be employed. We would like to stress that the resulting code can be compiled as efficiently as the original code.

Naturally, several well known techniques are used to support the basic ideas: symbol table, extension of procedure signatures, dependency analysis ... Instead of a formal treatment, the fundamental ideas shall be presented through examples.

2. Constant And Type Declarations

Constants and types declared in a procedure are taken out of the procedure and are declared as global entities. For example, see Figure 1 (the identifier X' is a renamed identifier X). This is an example that gives also a motivation for the approach. The translation process makes procedure `Q` a global procedure. Thus, both constant C_1 and type T_1 (modulo renaming) have to be declared as global entities.

3. Local Procedures

Let us recall that all local procedures are taken out of the procedure

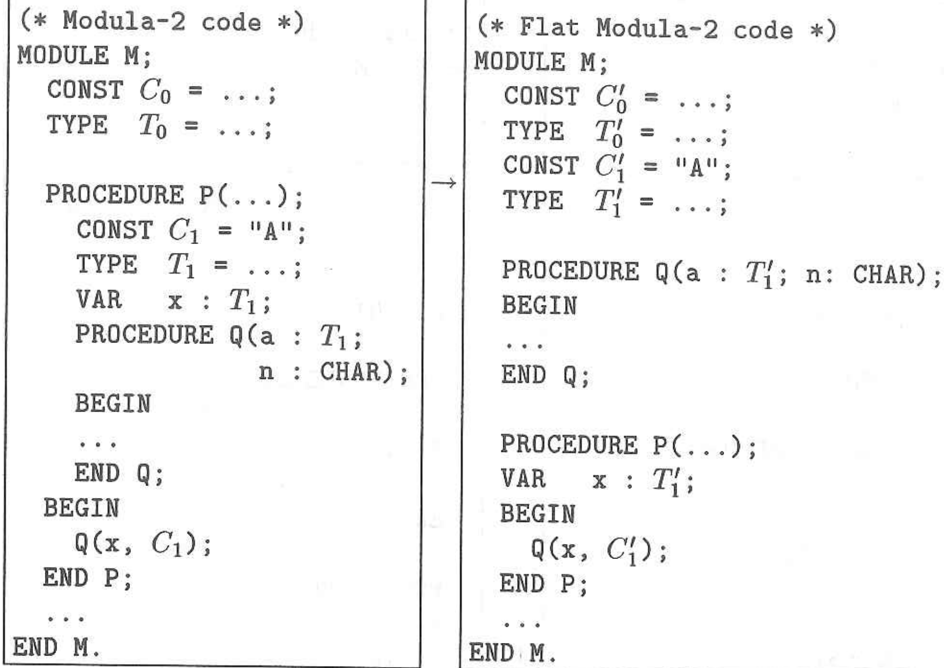

Fig. 1: Constant and type declarations

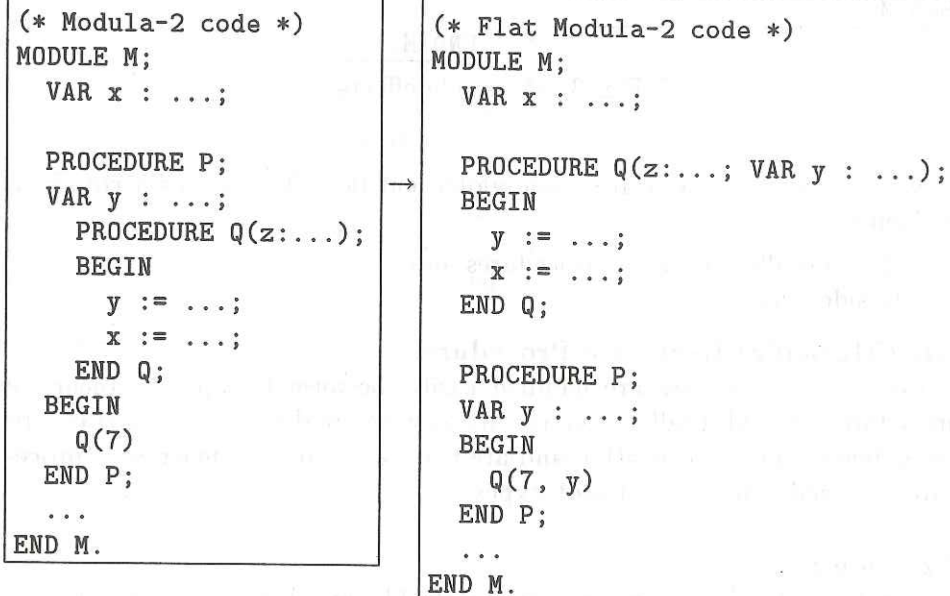

Fig. 2: Side effects

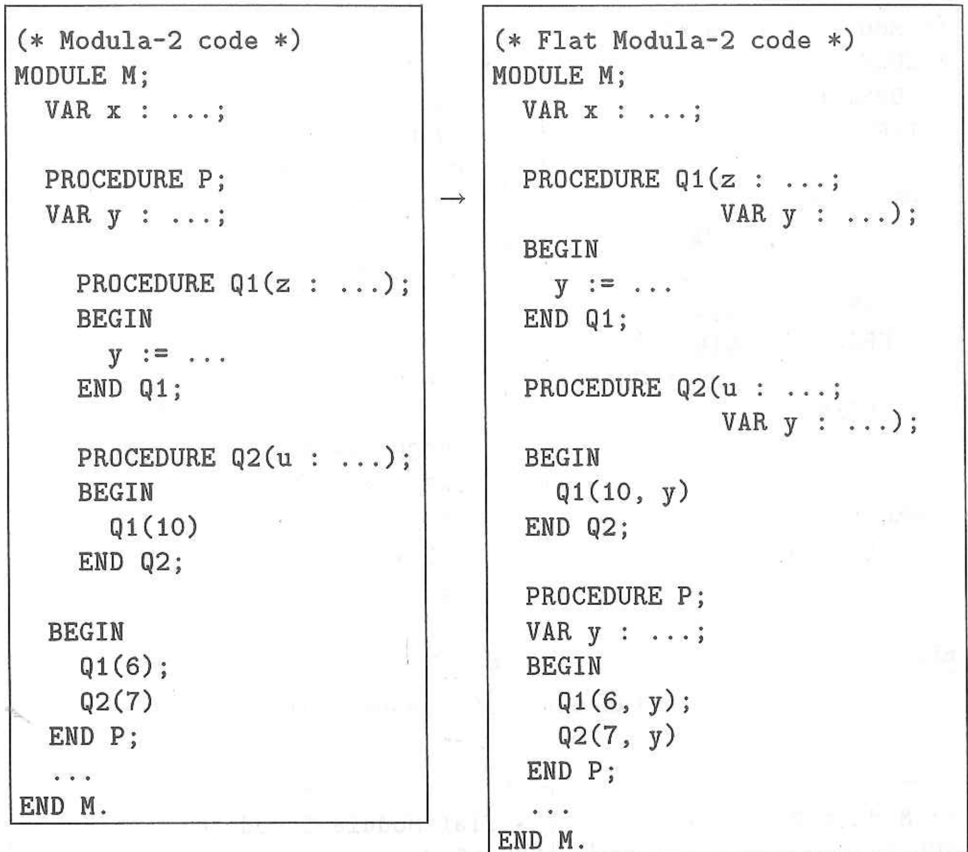

Fig. 3: More side effects

they are declared in and are made global entities. This raises a couple of problems:

(1) (mutually) recursive procedures and
(2) side effects.

3.1. (Mutually) Recursive Procedures.
Recursive procedures are handled easily, because C supports recursive procedure calls. Mutually recursive procedures are detected using standard dependency analysis algorithm and are translated to a sequence of C procedures preceeded by a set of prototypes.

3.2. Side Effects.
Local variables, of course, remain local. The problem arises when nested procedure uses local variable whose nesting level is less then or equal to

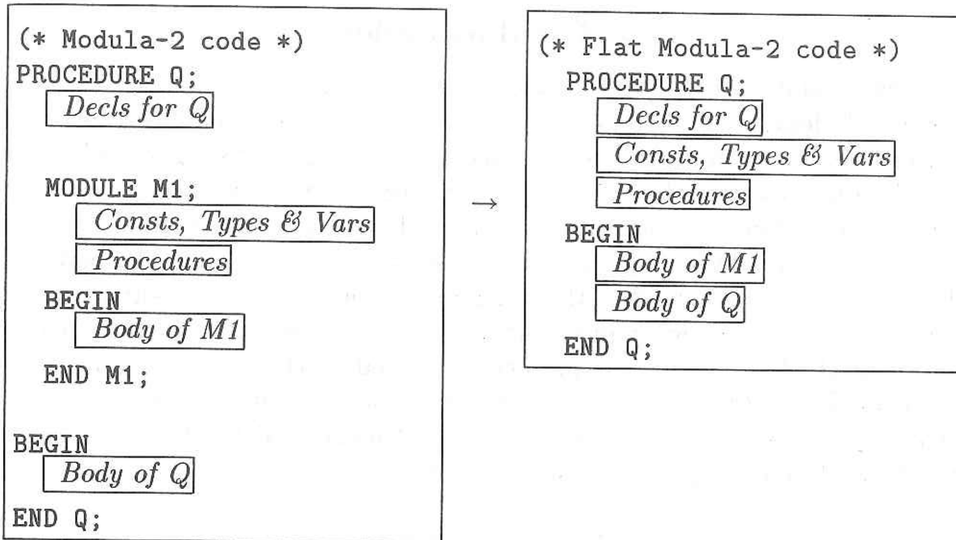

Fig. 4: Local modules

the nesting level of the procedure itself (this situation is known as a side effect). After globalization of a nested procedure, local variable declared in the surrounding procedure is no longer available to the globalized procedure. Consider an example given in Figure 2. Procedure `Q` changes the variable `y`. After globalization, procedure `Q` does not have access to variable `y`.

The best solution is to extend the signature of (previously) nested procedure and to pass the variables as `VAR` parameters. This change is recorded in symbol table in order to extend the signature in procedure calls as well. In our example this means that after globalization another formal parameter has to be introduced to procedure `Q`.

Unfortunately, the problem is not as simple as it has just been presented. There are situations in which a local procedure does not have side effects, but it depends upon other local procedures which do have side effects. Signatures of such local procedures have to be extended, too, in order to obtain correct translation. These situations are easily discovered (an unavoidable dependency analysis does the job), and are recorded in symbol table. As an example, consider the module and its translation given in Figure 3: although procedure `Q2` does not change variable `y`, it calls procedure `Q1` that does change the variable. This is why the signature of procedure `Q1` has to be extended, too.

4. Local Modules

Local modules serve only one purpose: to regulate the visibility and accessibility of identifiers. Systematic renaming and symbol table book-keeping during the translation process can take care of these tasks. Therefore, the translation of local modules is straightforward: the module bounds are broken, the identifiers renamed (having in mind the `IMPORT/EXPORT` lists) and the declarations are included in the surrounding environment. The body of the local module is moved to the beginning of the body of the surrounding entity (another module or procedure). Thus, the semantics of the initialization part of the module is preserved, as well as the initialization order. After all the local modules are removed, previous procedures can be applied to flatten the code. All these ideas are demonstrated in the example in Figure 4. It shows a procedure and its translation.

5. Conclusion

The paper has presented basic ideas upon which a translator of Modula-2 programs to C can be based. It has payed attention to translation of procedures and modules local to other procedures and modules, because other Modula-2 language constructs are easily translated to equivalent C constructs. Systematic renaming and globalization of local entities have appeared as key techniques in the process of translation.

The translation process requires two passes. In the first pass the symbol table has to be constructed and all the dependency analyses performed. The first pass can also break local modules and take care of renaming. After the first pass has been completed, the code can be generated in the second pass. Since all the checkings and analyses have been performed in the first pass, the second pass can be carried out very quickly.

Appendix A: Translating Flat Modula-2 to C

In this appendix some Flat Modula-2 programs are translated to a C equivalent just to give the reader a raw idea how the task can be performed.

```
(* Flat Modula-2 code *)
MODULE M;
  CONST C'0 = ...;
  TYPE  T'0 = ...;

  PROCEDURE P(...);
  BEGIN
    [B1]
  END P;
  ...
END M.
```

→

```
/* C code */
#define C'0     ...
typedef ... T'0;

void P(...) {
      [B1^C]
}

int main() { ... }
```

```
(* Flat Modula-2 code *)
MODULE M;
  VAR x : ...;

  PROCEDURE Q1(z:...; VAR y:...);
  BEGIN
    y := ...
  END Q1;

  PROCEDURE Q2(u:...; VAR y:...);
  BEGIN
    Q1(10, y)
  END Q2;

  PROCEDURE P;
  VAR y : ...;
  BEGIN
    Q1(6, y);
    Q2(7, y)
  END P;
  ...
END M.
```

→

```
/* C code */
... x;

void Q1(... z; ... *y) {
  *y = ...;
}

void Q2(... u; ... *y) {
  Q1(10, y)
}

void P(void) {
  ... y;

  Q1(6, &y);
  Q2(7, &y);
}

int main() { ... }
```

References

[1] N. Wirth, *Programming in Modula-2*, 4th Ed., Springer-Verlag, Berlin, 1988.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[3] N. Sundaresan, *Translation of Nested Pascal Routines to C*, SIGPLAN Notices, May 1990, pp. 69–81.

[4] M. Martin, *Entwurf und Implementierung eines Übersetzers von Modula-2 nach C*, Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, 1990.

Institute of Mathematics, University of Novi Sad, Trg Dositeja Obradovića 4, Novi Sad, Yugoslavia
E-mail address: ilehel@unsim.ns.ac.yu

Institute of Mathematics, University of Novi Sad, Trg Dositeja Obradovića 4, Novi Sad, Yugoslavia
E-mail address: masul@unsim.ns.ac.yu