IA

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 33/75 FEBRUARI

HENDRIK BOOM

THE ORGANIZATION OF THE OBJECT CODE GENERATOR IN ALGOL 68 H

2e boerhaavestraat 49 amsterdam

5751.849
BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A30

ACM -Computing Reviews- category: 4.12

The organization of the object code generator in Algol 68 H

Hendrik Boom, Mathematisch Centrum, Amsterdam

Abstract

The general approach to object code generation in the Algol 68 H compiler is presented. Algol 68 H is to be a compiler for all of Algol 68, written by one man in two or three years. The various intermediate languages used internally are explained, showing a gradual descent from a parse tree to a relocatable object module.

These languages are:

TREE: A parse tree, containing the essential information from the constructs of the Revised Report on the Algorithmic Language Algol 68. The parse tree canould be interpreted by an interpreter much resembling a Lisp interpreter.

STACK: This language is rather like postfix Polish, but with some non-postfix provisions for range entry, jumps, range exit, declarations, etc.

STORE: A language with conventional, possibly segmented, storage, and possibly a small amount of special storage (such as registers). In this language, values do not exist as independent entities on a stack. They always exist in storage, and are accessed by their storage locations. The heap and garbage collection are defined in this language.

REGISTER: This language is very similar to the store language, except that the nature of the special storage and the amounts of main storage required by various data types are known. The transition from the STORE language to the REGISTER language marks the introduction of machine dependence.

ASSEMBLER: An assembler language, with machine instructions and labels.

OBJECT: The relocatable object module, ready for linkediting or loading.

Although the author is now at the Mathematisch Centrum, the work reported here was done at the University of Alberta, Edmonton, Canada.

1. The Algol 68 H compiler

Algol 68 H is a compiler for Algol 68, at present written in Algol W. When complete, it will accept all of Algol 68 as defined by the Revised Report on the Algorithmic Language Algol 68 (except for bugs in the Report) and produce object code with complete run-time checking. Algol 68 H is largely machine-independent, so an attempt was made to make the object code generator machine indepenent whenever this did not cause inconvenience. It is intended that compatible versions of Algol 68 H will eventually run on a variety of machines.

Algol 68 code generation proceeds, at least conceptually, by taking the program to be compiled through a succession of versions, each written in a different intermediate language. Before discussing the transformations necessary to convert text from one version to the next, let us discuss just what these versions are.

2. The intermediate languages

2.1. First language, TREE

A program in this language is the output from the coercion phase, and the input for the object code generator. It is a parse tree with all necessary coercions marked. It contains that information from the "production trees" of the Report which is needed for object code generation, but in a more compact form.

Specifically, the tree consists of a conventional data structure connected with pointers. The LAYERs of the Report are not specifically represented, but each applied occurrence of an indicator has a pointer to its defining occurrence. Modes are represented in a graph of all modes, which contains only one canonical representative of each equivalence class of equivalent modes.

This data structure could conveniently be interpreted by a recursive interpreter, thereby producing a rather slow Algol 68 implementation.

2.2. Second language, STACK

This language strongly resembles Polish postfix notation. However, there are some unusual operations which do not fit into normal postfix notation. In particular, there are labels, conditional and unconditional jumps, range entry and exit operations, and ascription.

The flow of control in the stack machine is synchronized with the stack in such a manner that the number and size of the stack cells required for a routine at each point in its execution can be determined statically, except for the elements of arrays.

The STACK language and the TREE language are value-oriented. Values are obtained because they are possessed by identifiers, yielded by constructs, appear on the stack, etc. No consideration is given to the possibility that values might actually reside in storage. Even variables are not treated as if they were storage, but instead, as in the Report, they are mysterious "names" which mysteriously "refer" to other values.

In a straightforward implementation of this language, an enormous amount of copying will occur, especially when arrays are placed on the stack.

2.2.1. Typical operations

normal arithmetic.
assignment.
identity relation.
disuniting to extract the union flag.
disuniting to extract the value.
jump.
call.
fetch value possessed by identifier onto the stack.
fetch constant value onto the stack.
voiding.
subscripting.
ascribe value to identifier.
...

2.2.2. Operands

Operands are placed onto the stack by the two fetch operations. Except for this, operands of all operations come from the top of the stac, and yielded values are placed there.

2.3. Third language, STORE

This language deals with storage. Operations are expressed as operations on the contents of storage locations, and values exist only as contents of storage locations. Values are not treated as independent objects, but as slaves that must be carefully kept in their places, subservient to storage locations. It is in the STORE machine that the notion of "copying" values becomes significant. In the earlier languages, values did not have "instances" and therefore copying was meaningless. In this language, all values occupy storage (with one exception), and therefore proper copying is very important.

The exception mentioned above is that there may exist "literals", i.e., constants within instructions. A literal may be considered as a value which occupies storage in the instruction stream, rather than occupying the usual dynamically allocated storage; it may also be considered as an extension to the operation code of an instruction. Each approach is taken from time to time.

Very little is assumed about the nature of storage. In particular, nothing is known about the quantum size of addressible storage or about boundary alignment of values. When a request for allocation of a particular piece of storage is issued, it is of the form:

"Give me space to keep 13 <u>long real</u> values",

rather than of the form:

"Give me 26 words of storage".

Furthermore, it is assumed that storage may be paged in very small pages, with the page management performed at compile time. Therefore, whenever it is of particular importance that a value reside in store, an operation "ensure in store" is used. Nothing is assumed about the extra paging store.

2.3.1. <u>Operands</u>

Operands are of several forms:

1. Literals — a denotation provides the value required.

2. Directly addressible storage location.
   It is assumed that there is at least one directly addressable storage location. It is used to keep a pointer to the current (top) stack segment.

3. Indirectly addressable storage location.
   This is in the form of a base value, which is an operand, to which is added a displacement value. The resulting address addresses a storage location where the value is found. The displacement, although it represents an integer, is kept in symbolic form. Therefore, programs written in this language do not need to know how much storage is required for data values.

4. Squashed address.
   This is just like the third form, except that the resulting address <u>is</u> the value, and does not address it. This form is often used for variables defined by declarations like:

   <u>real</u> x, y, <u>int</u> z

2.3.2 Typical operations

- normal arithmetic operations.
- copy a value from one location to another.
- no assignment. Assignment (especially for arrays) is a complicated operation on values, which may involve one or more copy operation.
- Arrays are no longer present as such. Descriptors do exist. They are structures whose constituent fields are upper bounds, pointers to elements, etc. An array assignment from the STACK language becomes an appropriate loop of copy operations and storage allocations in the STORE language.
- Storage allocation operations are of the form:

  "Get me enough storage for 17 values of mode complex".

Storage may be "static", in which case it is allocated at compile time in the local stack segment, "local", in which case it is allocated on the dynamic stack, or "global", in which case it is allocated on the heap.

2.4. Fourth language, STACK

In this language, many details of the hardware machine are permitted to show through. In particular, the number of addressable units required for values, the boundary alignment required for values in store, and the nature of the machine's registers are known. Most concrete machines have special registers with special properties. Often operations are possible only when the operands are in a special arrangement in storage and registers. The register language reflects these peculiarities.

Registers are treated as paging memory. The attitude is that values belong in storage, but may be shoved into registers when that is necessary. When values are in registers, they stay there until for one reason or another they need to be put away into storage. One register is permanently reserved to hold a pointer to the current local stack segment.

The distinctive properties of registers and storage are the following:

Registers	Storage
in short supply	unbounded supply, for practical purposes.
directly addressible	only indirectly addressible.
capable only of containing "small" data values, such as pointers, integers, and reals, without scope indication.	capable of containing any value

The same registers are shared by all procedures; they must be saved and restored.	Each call to a procedure has its own storage, which is not shared.

Programs written in the REGISTER language are extremely complicated affairs, full of macro-like conditional assembly. They contain compile-time tests on the presence of values in registers and store, and corresponding sets of instructions to be inserted conditionally, depending on the results of the tests.

2.5. Fifth machine, ASSEMBLE

This is, except for notational differences, an assembly language. There should remain only the counting out of machine instructions and placement of branch addresses. On the IBM 360 and some minicomputers, there remains also the replacement of instructions containing large displacement values by lawful instructions without such large displacements. Since only the assembler knows the addresses of machine instructions, only the assembler can deal with branches apparently requiring large displacements.

3. Representation of intermediate texts

Many of the translations from one language to another can be done by straightforward one-pass processes. It is foolish, therefore, to perform these translations one at a time, since this would require that each of the intermediate texts of a program be kept in storage in its entirety between passes. Instead, the translations are done in parallel. As soon as one translator produces a piece of output, it is passed to the next translator, which acts as necessary, possibly producing input to yet another translator. It is therefore necessary to keep only a very small amount of intermediate text at any one time. Such intermediate text, which is consumed as fast as it is produced, is called "virtual" text, since the entire text never actually exists at any one time.

The code in the languages STACK, STORE, and REGISTER exist as virtual text. The text for TREE and ASSEMBLER is actual. However, the assembler processes a single procedure at a time, so that only the code for a single procedure need be kept in its entirety at any one time.

The information in each instruction is passed to the corresponding translator by a procedure call. The operands are the parameters of the procedure, and the operators are in one-to-one correspondence with the procedures available to be called.

The STACK language is not visible in the object code generator for the following reason. The run-time stack for the STACK language has a compile-time analogue, which is kept in the stack used by the compiler itself for its own procedure calling. Since this stack is maintained implicitly by the language in which the compiler is written, no explicit code for the STACK machine appears in the compiler.

The extremely complicated conditional assembly mentioned for the REGISTER machine is also virtual. Rather than creating tricky texts and then interpreting them, the required condition tests and actions are simply performed.

The intermediate texts which it might be convenient to make actual instead of virtual are TREE, STACK, STORE, and ASSEMBLER. Only TREE and ASSEMBLER are actual in the present version of Algol 68 H. If others were actual, it would be possible to use multiple pass algorithms to perform significant program optimization. Common subexpression elimination could be performed on TREE code or perhaps on STACK code. Clever register allocation, depending on anticipating future use, could be performed in the translator from STORE to ASSEMBLER. In addition, more unnecessary copy operations might be eliminated in the STORE machine.

4. Data structures used in effecting the transformations

The object code generator attempts to delay execution of certain operations whenever it can. This often enables them to be combined with other operations, performed at compile time, or suppressed entirely. These operations are field selection, constant arithmetic, and dereferencing and other forms of copying. Many concrete machines, for example, have a single instruction which is capable of compiling a dereferencing, a field selection, and a multiplication.

Operations on real numbers are never modified or performed at compile time. Not even the associative law can be relied on. Furthermore, the machine on which a program is compiled may be different from that on which it is run. This might be enough to make real arithmetic at compile time radically different from that at run time.

The compile time data structures are so designed as to make these optimizations easy.

Run-time objects, such as values, storage locations, and stacks, are represented at compile time by "models". The prefix "m" is used to represent models. Therefore, models of values, store, and the stack are called "mvalues", "mstore", and the "mstack". The phoneme "m", being a semivowel, may be pronounced as a separate syllable.

The TREE language gets "mvalues".
The STACK language gets "mvalues" and an "mstack".
The STORE language gets "mstorage".
The REGISTER language gets "mregisters".

Relationships exist between the models for each language and the models for the next language. These relationships are often encoded using pointers. Therefore, for example, the model of a value points to the model of its storage location. At run time, the value will itself reside in that storage location.

Let us discuss these relationships in some detail.

The mstack points to mvalues. One same mvalue may in fact appear at several different positions in the mstack, corresponding to the yields of different constructs.

Mvalues are meaningful in the TREE and STACK machines. Since it is sometimes necessary to process all currently existing values (for example, at garbage collection time), all these mvalues are linked into a single chain. This chain is not the mstack, since mvalues sometimes behave in a nonstacklike fashion. For example, upon exit from a serial clause delivering a value, the top entry remains on the stack, but values possessed by identifiers vanish.

If the value represented by an mvalue is known at compile time, that value is a part of the mvalue.

If a value is in storage, the mvalue points to the mstorage.

If the value is known to be one of the entries in the "display", which points to the stack segments for other currently known ranges, this is indicated in the mvalue.

An mstore contains an integral displacement and a pointer to a base mvalue. At run time, the address of the storage can be computed by adding the displacement to the corresponding base mvalue. Storage is always addressed via a base mvalue, in order to preserve re-entrancy of the object code. The base value is often a display element, but it is sometimes a reference value or the pointer occurring in an array descriptor. This allows dereferencing and field selection to be delayed, and eventually combined with other operations.

All mstores corresponding to storage in the most local stack segment are linked together, in order, for the convenience of the static storage allocator.

Mstores and mvalues contain reference counts, to permit release of compile-time and run-time resources when they are no longer needed.

If a value supposed to reside in a storage location is in a register, the mstore points to the mregister, and the mregister points to the mstore.

An mstore contains an indication whether its contained value actually resides in store, in a register, or both.

An mstore contains an indication of whether it is read-only. For example, the mstore corresponding to 'i' in

<u>int</u> i = 3+j

is read-only. Mvalues and mstore with a reference count greater than one are also treated as read-only, until the reference count becomes one or zero again.

Registers may be "locked" temporarily. A locked register is not available for allocation by the register allocator. An unlocked register is available for allocation; however, if it contains a value, it may have to be stored into storage before such allocation. Certain registers, such as the one containing a pointer to the local stack segment, are <u>always</u> locked.

5. Degree of machine dependence

Preliminary experience on the IBM 360 indicates that only about half of the object code generator is machine dependent. This is smaller than was originally expected. It is quite likely that further thought can reduce this fraction further.