Using C as Target Code for Translating Highlevel Programming Languages

Pre–publication version — DO NOT COPY OR DISTRIBUTE

Ulrich Hoffmann

Christian-Albrechts-University of Kiel

Task: Compilation to C
Status: Draft
Document Identifier: APPLY/CAU/IV.4/2
Created: 8.11.93
Revised: 18.11.93

© 1993 CAU

Synopsis

It is now more than ever necessary to develop system level software portably because of the rapid changes in hardware and the already existing variety of platforms. This article will focus on compilers which translate from *highlevel* programming languages. They can be distinguished from *higher level* languages in that highlevel languages also include non imperative language features and dynamic data structures. When moving a highlevel language compiler to a new platform the implementor has to choose between a tedious implementation with maximal efficiency on one hand or a portable but less efficient implementation which uses a machine independent target language on the other hand.

Recently out of this conflict the idea evolved to use C as machine independent target code. Typically there already exists an optimizing compiler for standard C on the machine in focus, so a portable implementation with acceptable performance is possible. In addition to this compiling to C makes it easier to integrate highlevel programs with other software systems.

Existing work in this area normally follows an ad hoc approach to translate to a sublanguage of C, which is not described precisely. It remains open how efficient C compilers can translate this sublanguage to machine code.

This article develops the view of an abstract C-machine to describe translations to C. This allows to systematically examine C as machine independent and efficiently compileable target language.

The first part will give an overview how to compile via abstract machines in general and will also address existing work on compiling to C. The second part will bring these to approaches together. The C-machine as used in the joint project APPLY to compile CommonLisp to C is explained.

Finally, we will introduce some of the principle problems compiling highlevel languages to C.

This work was supported by the German Federal Ministry for Research and Technology (BMFT) within the joint project APPLY. The partners in this project are the Christian

Albrechts University of Kiel, the Fraunhofer Institute for Software Engineering and Systems Engineering (ISST), the German National Research Center for Computer Science (GMD), and VW-Gedas.

Christian Albrechts University Kiel
Institute for Computer Science and Practical Mathematics
Research Group "Programming Languages and Compiler Construction"
Preusserstr. 1-9, W–2300 Kiel 1, FRG
Fax: 0431/5661-43
Person to contact: Dr. Friedemann Simon
Phone: 0431/5604-25
E-mail: fs@informatik.uni-kiel.dbp.de

German National Research Center for Computer Science (GMD)
Institute for Applied Information Technology
Artifical Intelligence Research Division
P.O. Box 1316, W–5205 Sankt Augustin 1, FRG
Fax: 02241/14-2072
Person to contact: Prof. Dr. Thomas Christaller
Phone: 02241/14-2678
E-mail: thomas.christaller@gmd.de

Fraunhofer Institute for Software Engineering and Systems Engineering (ISST)
Kurstr. 33, O–1086 Berlin, FRG
Fax: 030/20372-301
Person to contact: Dr. Horst Friedrich
Phone: 030/20372-342
E-mail: horst.friedrich@isst.fhg.de

VW-GEDAS
Pascalstr. 11, W–1000 Berlin 10, FRG
Fax: 030/39007999
Person to contact: Dipl.-Ing. Jürgen Beste
Phone: 030/39007-331
E-mail: beste%sun1@hwsw.gedas.de

Contents

1 Introduction

During the last years hardware developed in a stormy fashion. This makes it hard for software, both application software as well as system level software, to follow. To support transfers of programming language research results to industry it is important that program development tools and the programs they generate match industrial environments. Highlevel programming languages, i. e. languages whith continuations and higher order functions, automatic memory management and dynamic data structures on a high level of abstraction, are the basis of current programming language research.[1] In contrast, these languages are not (yet) widely adopted in industry.

Today, the programing language C, because of its exposed position in UNIX, forms a portable and efficient implementation plattform, which spans a broad variety of hardware. This immediately rises the idea to use C as a machine independent target language for compilers of highlevel programming languages. Programs generated this way are portable, yet efficient and can be easily integrated to other software systems. The normal approach compiles to an unprecise sublanguage of C. This prohibits systematic examination of these translations.

A standard technique to implement an arbitrary compiler follows the path via generating code of an abstract machine, which is optimally adopted to the specific translation requirements. This paper tries to join these ideas, compiling via abstract machines on one hand and to compile highlevel languages to C on the other hand. By constructing abstract C-machines it will be possible to precisely describe the translation from highlevel languages to C and so to get an in depth knowledge of its pros and cons.

2 Abstract Machines and Translation

The classical approach to translate programming languages uses code of an abstract machine as its target language. This technique is shown in figure 1.

Historically the main reasons in doing so were limited resources and the good portability of generated code. By implementing an abstract code interpreter it was comparibly easy to build the same runtime environment for abstract code on numerous platorms. Components of an abstract machine were typically designed for the specific needs of executing a given language. Because of this, abstract code tends to be much shorter than corresponding native machine code.

Using an interpreter for abstract machine code lead to much longer execution times than executing real machine code. To circumvent this limitation simple native code generators translated abstract machine code more or less by macro expansion with peephole optimization to real machine code. Since they normally did not perfom any structure modifying optimizations, executing the resulting machine code was still slower than directly generated native code. In detail, the way of a classical translator with abstract machine code looks like this: First the target language is parsed and its static semantics is checked. Then optimizers transform the resulting internal representation and finally the code generator

[1] In this respect, the higher level languages C and Pascal do not qualify as highlevel language, whereas Lisp, Scheme or ML do.

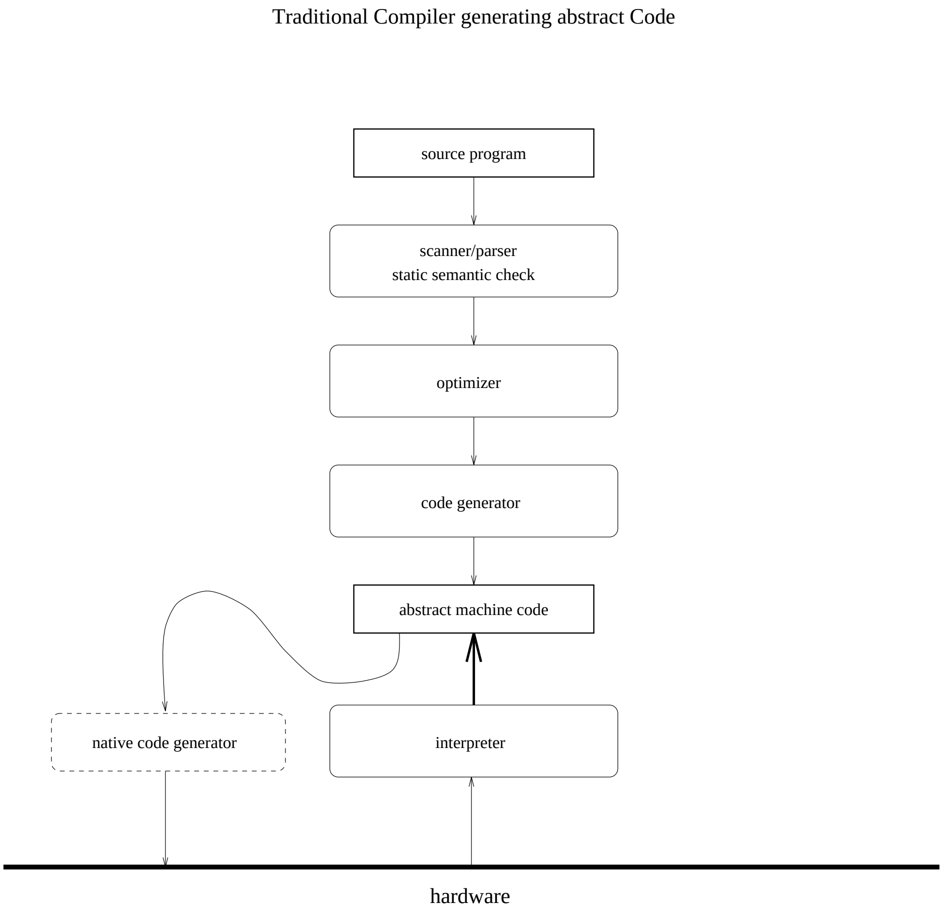

Figure 1:

produces abstract machine code. An interpreter can execute this abstract machine code, or as described above, a simple native code generator can produce native code for direct execution by the underlying (concrete) machine.

The following paragraphs introduce shortly some classical implementations which used abstract machine code as target language.

An important criterion of abstract machines is the existance of one or more run time stacks which hold activation records in on or the other form. Calculations are performed in either registers or directly on the stack.

2.1 Classical imperative programming languages

The term *classical imperative programming languages* describes programming languages having the assignment as their main feature. This term is necessary to distinguish between them and object-oriented languages.[2]

[2] In general, a clear distinction of programing languages — especially in case of hybrid languages — is hard and is always result of the current point of argumentation.

2.1.1 BCPL

BCPL is a comparitively simple classical imperative language. Its only data type is the machine word. M. Richards introduced a portable implementation of BCPL around 1970 [Ric71]. It uses two different abstract machine languages — OCODE (*intermediate Object Code*) as target language in the sense described above and INTCODE (*interpretive machine code*) which was specifically designed to port the compiler to new platforms by bootstrapping. For this INTCODE has only 8 different instructions (arithmetic counted as one). An interpreter for INTCODE was very easy to implement, but the resulting performance was low. On the other hand it was more difficult to implement an OCODE interpreter but this resulted in shorter execution times. Both, OCODE and INTCODE, have a stack to hold control information and arguments intermixed. Arithmetic in OCODE is performed on top of the stack. INTCODE uses two special purpose registers for this.

2.1.2 Pascal

Pascal was developed by N. Wirth at the end of the 1960s [Wir71] for educational purposes. The prototype compiler for Pascal, P4, generates PCODE (*Pascal code*) [PD82], code for an abstract machine. A complete development system for Pascal, the UCSD-P-system, was developed at the University of California, San Diego (UCSD). It became very popular as a programm development system for the APPLE II micro computer and was also available on several other platforms. The whole system was written in Pascal and so could be compiled to PCODE. Since all systems were binary compatible at the PCODE level, it was very easy to port applications from platform to platform. The abstract Pascal machine had a stack which held data as well as control information (static and dynamic chains). Calculations were perfomed on the stack.

2.1.3 Modula 2

As for its predecessor, Pascal, there is also a Modula 2 compiler to abstract machine code (MCODE). A special purpose processor in the Lilith workstation [Jac82] [Wir81] was especially designed to execute MCODE directly. C. Jacobi describes the semantics of MCODE by presenting an MCODE interpreter written in Modula 2.

2.2 Objekt oriented programming languages

2.2.1 Smalltalk-80

Smalltalk-80 [GR83] is widely considered to be the classical object oriented programming language. In the sense of this article Smalltalk qualifies as highlevel programming language, because it provides higher order functions (blocks) and needs automatic memory management. The semantics of Smalltalk, e. g. method lookup, is defined via an abstract Smalltalk machine, which executes so called BYTE code. Its semantics on the other hand is determined by an interpreter written in high level Smalltalk. The abstract Smalltalk machine holds arguments, control information and performs calculations on a single stack. The Smalltalk implementation done by Deutsch [DS84] incrementally transforms abstract machine code to concrete machine code at run time, which is executed on the fly.

2.3 Funktional programming languages

2.3.1 Lambda calculus: SECD machine

The SECD machine, introduced by P. Landin [Lan63], provides an extremely simple execution model for the λ-calculus. By using simple rules it evaluates λ-terms in an applicative order startegy (call by value). The SECD machine has 4 stacks, stack S for calculating intermediate results, stack E (*environment*) to represent the current variable bindings, stack C (*Control*) to hold the expression to be evaluated and stack D (*Dump*), to store the states of the other stacks during evaluation of function applications. In its original pure form, the SECD machine is not well suited for efficient execution of functional programs, because it operates directly on the textual representation of λ-terms. There are, however, numerous variations of the SECD machine (e. g. for Hope [BMS81]), which were used for efficient implementations of functional programming languages.

2.3.2 Lazy ML: G-machine

In 1984 J. Johnsson and L. Augustsson constructed the G-machine [Aug84] for compiling Lazy ML. In principle it is also a variation of the SECD machine and could as well be called SGCD machine. During program execution the G-machine transforms a graph representation of the program. The reduced graph is the result of this calculation. On its stack S, the G-machine passes arguments and performs calulations. It constructs new expressions in the graph G for partial unevaluated intermediated expressions. Initially G-machine program code is located in the program graph and is loaded successively into the instruction cache C for execution. Stack D is used to save execution contexts, when nested calculations take place. G-machine code is optimized in several ways and is finally macro expanded to assembly code of the underlying target machine.

2.3.3 Haskell: Spineless tagless G-machine

The so called *spineless tagless G-machine* evolved in Glasgow at the end of the 1980s [PJS89] to compile Haskell programs efficiently on standard hardware. It tries to eleminate the weak points in the G-machine design: traversing the graph to determine function arguments and dispatching within G-machine instructions depending on argument types. Spineless tagless G-machine code is transformed to C code. This will be described in section 3.3.

2.4 Logic oriented programming languages

2.4.1 PROLOG: WAM

In 1980 D. Warren developed an abstract machine, WAM (*Warren Abstract Machine*), to execute PROLOG progams. Today numerous WAM variations build the foundation of standard PROLOG implementations. The WAM has program and data memory, 3 stacks as well as several registers. It has very complex instructions. While executing PROLOG programs, the WAM components cooperate tightly coupled. Control information is spread on the stacks. Calculations take place in argument registers via a special instruction.

3 Generating C target code

It is normally a large effort to implement a compiler for a highlevel language. Thus it is reasonable to make the compiler as portable as possible, so it can be transfered to the next generation of computers without problems. Nevertheless it is desirable that the generated code can be executed as effiecient as code produced by a nonportable, but machine specific compiler.

Most parts of modern operating systems are implemented in C. Because of this, the first step to port such an operating system to a new platform is to implement a C compiler on the new machine. After that major parts of the operating system and most of the application programs can be ported by recompilation.

It makes sense to do the same for compilers. For some imperative programming languages (Fortran, Pascal) there are compilers which transform programs to C. This is without problems, as long as C language features coincide with those of the source language. Highlevel languages, however, have language features not found in C. These must be simulated by available constructs, which may lead to loss in efficiency. From existing work [TAL91] we know, that programs, if compiled to C, have an execution time which is in the same range as programs generated by the best implementations (factor 2 in case of ML). Thus, compiling via C leads to better results, than building an ad hoc native code generator for the new machine.

In addition, going via C opens the possibility for good cooperation of compiled programs with other standard software like numerical C-libraries or user interfaces. This advantage is very important. For native code compilers, the integration with standard software is tedious and must normally be implemented at high cost on each new machine.

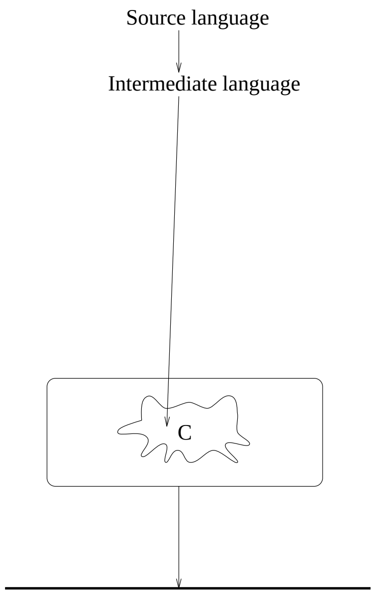

Figure 2:

The compilers introduced in the next sections are compilers in a classical sense and no interactive nor incremental programming environments, like those often found for highlevel programming languages. Each of theses compilers translates to a sublanguage of C, but its margin is not sharply defined. Only the system implementors know in detail which C language constructs are generated and to what degree the generated C code is restricted to one class of computers. So it is only possible to make statements about portability on a pragmatic level.

3.1 Scheme–>C

In 1989 J. Bartlet introduced a compiler for Scheme (a Lisp dialect) to C [Bar89] which can be considered the first well known and succesful attempt to compile a highlevel programming language to C. Scheme–>C compiles Scheme function definitions to corresponding C function definitions. If necessary local Scheme function are globalized, since C does not know about nested functions. Normal C implementations are based on a runtime stack, but it is not possible to access this stack explicitly. The design decision to compile Scheme functions to C functions and the inaccessability of the C run time stack make it impossible for Scheme–>C to fulfill the complete elimination of tail recursions as required by the Scheme language definition. Instead Scheme–>C performs a static analysis to detect and eliminate at least simple cases of tail recursions. Scheme–>C passes Scheme arguments as C paramters so Scheme objects are stored in the C runtime stack and, depending on the C compiler, in machine registers. To do memory management under these conditions, J. Bartlet developed a new memory management strategy, which avoids moving certain objects when memory is cleaned. In addition the Scheme–>C runtime system contains some machine- and C compiler-dependant parts written in assembler to implement continuations, to traverse the C runtime stack and to inspect references to Scheme objects.

3.2 ML to C

Starting with the results of Scheme–>C a compiler from ML to C was developed at Carnegie Mellon University [TAL91]. The runtime requirements of ML are similar to those of Scheme, so the implementation technique of `ML to C` can be used as well to implement Scheme. The translation performed by `ML to C` is based on a continuation passing style to handle ML exceptions, continuations and complete eliminiation of tail recursions. C does not provide branch targets as first class data. Computed jumps can only be done by calling functions via function pointers. This increases the C runtime stack. Since C compilers normally do not perform tail recursion elimintation, `ML to C` translates ML function definitions according to the following schema: Instead of translating each ML function into exactly one C function, each extended basic block (they contain only forward jumps) is translated to a C function. The resulting C functions are connected by a mini interpreter[3]. First, this interpreter calls a translated extended basic block B_1. If this block needs to transfer control to another basic block B_2, it returns the address of B_2 as function result to the mini interpreter, which actually performs the transfer to B_2. By this the C runtime stack is used only in a restricted way and branch targets become first class data which can be stored on a runtime stack of their own.

[3]This interpreter originates in an idea of G. Steele in his rabbit compiler, where it was called UUO-handler. In `ML to C` it is called the *apply like function*.

3.3 Haskell

The Haskell compiler developed in Glasgow compiles programs in a first step to *spineless tagless G-Machine* code [PJS89] and finally to C. Like `ML to C` it uses the mini interpreter technique, whereas the interpreter is accessed via C macros. Normally these macros expand to portable C code which handles the communication with the interpreter. For a specific machine however, these macros can be modified to expand to machine specific inline assembly code to shorten execution times of resulting programs. Because the structure of generated C code is known to be simple specific optimization supported by some C compilers can be used.

Although the Haskell compiler is based on an abstract machine, the sublanguage of C, which is used as target language, remains unspecified.

4 Abstract C-Machines

We will now link the idea to compile via abstract machines and the idea to use C as target code. This results in the conception of an abstract C-machine. By defining an abstract C-machine it is possible to describe the generated C code in much more detail than currently possible. The structure of a compiler which follows this idea is shown in figure 3.

There is no formal semantics definition for C yet. Thus it is not possible to systematically examine C code generation of existing compilation approaches. The meaning of abstract C-machine code however can be defined precisely by well known semantics definition techniques. So, it becomes possible to examine C code generation on a well founded basis.

It is reasonable, that a large number of components of the abstract machine have direct equivalents in C. For them, the compilation to C is more or less a direct program transformation. For continuations or dynamic memory management however, no such direct correspondance can be found. Support from a runtime system must be used in this case. The missing correspondance is reflected in the design of the abstract C-machine and we can conclude about it.

In addition, we can examine other aspects, such as the ability to integrate generated programs with other software systems, if we add special components to the abstract machine for this purpose. For example, well defined suspension and restart operations of the C-machine will allow us to describe cooperation with event driven user interfaces.

In contrast to directly generate concrete machine code from abstract machine code, C-machines have the advantage, that there already exist code generatores on numerous platforms. Often they perform global structure changing optimizations.

5 The APPLY C-Machine

This section will describe the abstract C-machine which is used in the APPLY-CommonLisp-to-C-Compiler (CLiCC).

To give an impression how the generated code is structured, we will start with a small example:

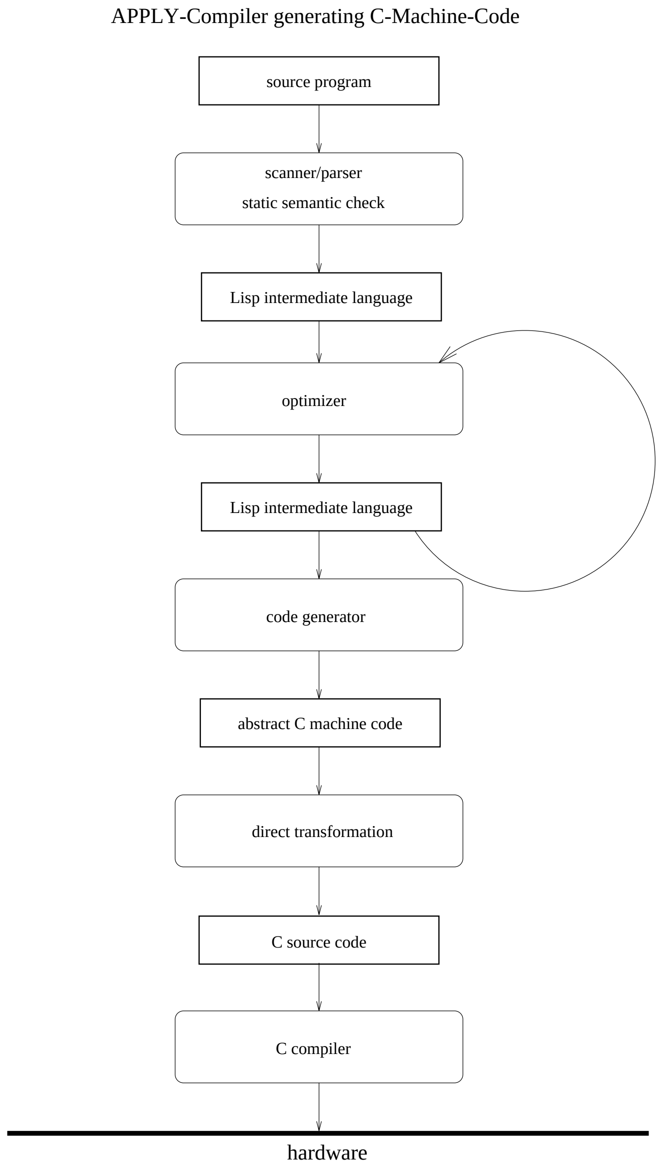
APPLY-Compiler generating C-Machine-Code
source program
scanner/parser
static semantic check
Lisp intermediate language
optimizer
Lisp intermediate language
code generator
abstract C machine code
direct transformation
C source code
C compiler
hardware

Figure 3:

5.1 Example Translation

```
(defun fac (n)
  (if (= 0 n)
      1
      (* (fac (- n 1)) n)))

(princ "Please enter a number: ")
(print (fac (read))) (terpri)
```

This program has a main program, which consists of the two calls to `print`. It queries the user to enter a number, calculates the factorial and finally outputs the result.

The function definition, without any optimizations, is translated to the following C code:

```
/* defun fac (n) */
void Ufac(CL_FORM *base) {

  /* if (= 0 n) */
  LOAD_FIXNUM(0, STACK(base, 1));
  COPY(STACK(base, 0), STACK(base, 2));
  Fnumeql(STACK(base, 1), 2);
  if(CL_TRUEP(STACK(base, 1)))  {
    /* 1 */
    LOAD_FIXNUM(1, STACK(base, 0));
  } else  {
    /*  (* (fac (- n 1)) n) */
    COPY(STACK(base, 0), STACK(base, 1));
    LOAD_FIXNUM(1, STACK(base, 2));
    Fminus(STACK(base, 1), 2);
    Ufac(STACK(base, 1));
    COPY(STACK(base, 0), STACK(base, 2));
    Fmult(STACK(base, 1), 2);
    COPY(STACK(base, 1), STACK(base, 0));
  }
}
```

Since there were no analyses nor optimizations, it has to be assumed that the function `fac` can be called with arguments of arbitrary types. Because of this, the parameters cannot be held in C variables nor handled as C parameters. However, a suitable optimization could support the code generator to produce an optimized function body.
The factorial function checks first, if the argument is equal to zero.[4] Intermediate results and function arguments are calculated on the Lisp stack. If the argument is zero, the function result is one. Otherwise the factorial function ist calculated in the well known recursive way.

The C code for the main program looks like this:

[4] The function `Fnumeql` would result in a runtime error, if called with non number arguments.

```
CL_INIT Klisp_main[] =
{
/* 0 */ { CL_FIXNUM, 4 },
/* 1 */ { RT_CHAR_PTR, (long)"USER" },
/* 2 */ { CL_SMSTR, (long)&Klisp_main[4] },
/* 3 */ { CL_NIL, (long)NIL_VALUE },
/* 4 */ { CL_FIXNUM, 4 },
/* 5 */ { RT_CHAR_PTR, (long)"LISP" },
/* 6 */ { CL_FIXNUM, 23 },
/* 7 */ { RT_CHAR_PTR, (long)"Please enter a number: " },
};

void lisp_main(CL_FORM *base) {
    ... /* Package Initialisierung */

    /* (princ "Please enter a number: ")  */
    LOAD_SMSTR((CL_FORM *)&Klisp_main[6], STACK(base, 0));
    Fprinc(STACK(base, 0), 1);

    /* (print (fac (read))) (terpri) */
    Fread(STACK(base, 0), 0);
    Ufac(STACK(base, 0));
    Fprint(STACK(base, 0), 1);
    Fterpri(STACK(base, 0), 0);
}
```

The array `Klisp_main` contains constant data, which is used by the main program, like names of packages or the prompt text from the source program.

First, the main program calls the function `Fprinc` to output the prompt text. Then a Lisp object is read by `Fread`, the factorial is calculated and the result displayed. Lisp functions, without a fixed number of arguments, are called, in addition to the start of the argument frame, with the number of arguments. The factorial function however has exactly one argument and thus the number of arguments is given implicitely.

5.2 Architecture of the APPLY C-Machine

The architecture of the APPLY C-machine is shown in figure 4.

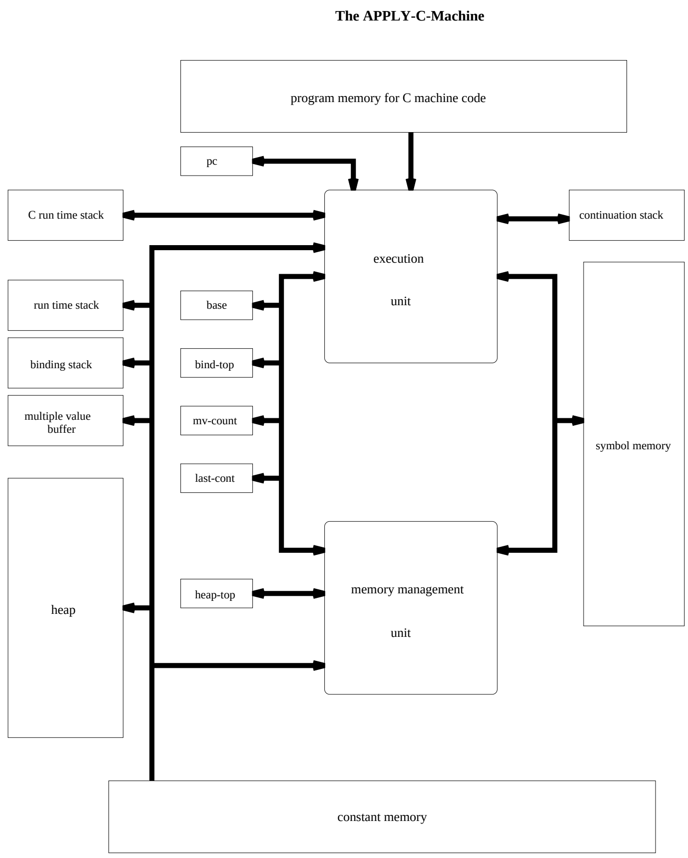

Figure 4:

The core of the C-machine is separated into two parts. The excution unit performs calculations and handles the execution of instructions. The memory management unit handles memory requests and cleans up memory if necessary. Both core units are connected by busses to different kinds of memory. Each bus handles different kinds of data. The symbol memory is connected by the symbol bus to the execution and memory management unit. The heap, the Lisp runtime stack, the constant memory and the special stacks for multiple values and dynamic bindings are connected by the form bus to the core units. Registers are accessible via the addresses bus. The execution unit can reach program memory by the unidirectional instruction bus. This shows, that program memory is considered read only. The C-stack can only be accessed by the execution unit. No data which has to be handled by the memory management unit is allowed to be on the C-stack.

5.3 Specifying the abstract Machine in Z

The behaviour of the abstract machine will be describes by state transitions, which take place when single instructions are executed. Inspired by a description of a concrete micro processor in Z by J. Bowen [Bow89] we will also use Z to exemplarily describe some aspects of the abstract C-machine. First, valid states are charactized. Based on this we will describe state transitions.

In case of the APPLY-CommonLisp compiler, the abstract C-machine operates on Lisp data. We will not describe this here in detail.

$[CL\_FORM]$

Single elements of CL_FORM[5] can be distinguished, but their fine struktur is not known.

The abstract machine can execute instructions and it has registers:

$$\begin{array}{rcl} \mathit{Instruction} & ::= & \mathit{LOADFIXNUM} \mid \mathit{CALL} \mid \mathit{RET} \mid \\ & & \mathit{STACKSTACKCOPY} \mid \mathit{LOADSMSTR} \mid \ldots \\ \\ \mathit{Regs} & ::= & \mathit{PC} \mid \mathit{BASE} \mid \mathit{BIND\_TOP} \mid \mathit{MVCOUNT} \mid \\ & & \mathit{LAST\_CONT} \mid \mathit{HEAPTOP} \mid \ldots \end{array}$$

We will regard only parts of the possible instruction set. A complete list will be much longer. Adressing modes, which appear in the concrete implementation, are converted to several instructions here for the sake of simplicity. For example a `COPY(STACK(base, . ), STACK(base, . ))` (see C-Code in 5.1) is modelled by the instruction `STACKSTACKCOPY`.

The registers have the following meaning:

- `PC`
  The program counter `PC` addresses the current instruction in program memory.
- `BASE`
  determines the position of the topmost stack frame on the Lisp stack.
- `BIND_TOP`
  determines the topmost element on the bind stack, which holds dynamically bound variables.
- `MVCOUNT`
  determines the number of Lisp objects in the multiple value buffer. This is used for Lisp functions that return more than one result.
- `LAST_CONT`
  determines the topmost element in the continuation stack.
- `HEAPTOP`
  determines the start of free memory.

[5] The abbreviation CL_FORM stands for CommonLisp-form.

The following Z schema describes the abstract machine state:

$$
\begin{array}{l}
\textit{C\_Machine\_State} \\
\hline
\textit{C\_Code} : \mathrm{seq}(\textit{Instruction} \times \mathbb{N} \times \mathbb{N}) \\
\textit{Op} : \textit{Instruction} \\
\textit{arg}1, \textit{arg}2 : \mathbb{N} \\
\textit{Reg} : \textit{Regs} \rightarrow\!\!\!\!\!+ \mathbb{N} \\
\textit{C\_Stack} : \mathrm{seq}\, \mathbb{N} \\
\textit{Lispstack} : \mathbb{N} \rightarrow\!\!\!\!\!+ \textit{CL\_FORM} \\
\ldots \\
\hline
(\textit{Op}, \textit{arg}1, \textit{arg}2) = \textit{C\_Code}(\textit{Reg}(\textit{PC})) \\
\textit{Reg}(\textit{PC}) \leq \# \textit{C\_Code} \\
\ldots \\
\hline
\end{array}
$$

The abstract machine state is described by the following components:

- Program code (C_Code)
  This is a sequence of instructions with at most two arguments. For instructions with arity zero or one, the additional arguments are irrelevant.

- Instruction register (Op)
  It contains the current instruction to be executed.

- Argument registers ($arg1, arg2$)
  They contain the arguments of the current instruction.

- Register contents (Reg)
  This partial function assigns each register a value.

- Stack for control information (C_Stack)
  The C-stack is modelled as a sequence of primitive values. This shows that it will be accessed only in a restricted way.

- Stack for intermediate results and arguments ($Lispstack$)
  Since it is necessary to random access the *Lispstack*, it is modelled as a partial function from indices to the set of Lisp forms. By this, the Lisp stack can be changed at arbitrary locations.

The state invariants determine that the current instruction, consisting of $Op, arg1$, and $arg2$ is addressed via the program counter `PC` in program memory, and that `PC` always is within the bounds of program memory.

If an abstract instruction is executed, it changes the abstract machine state. If nothing different is specified by the operation, the program counter will be incremented by one and the Lisp stack will be unchanged. Program memory is read only:

$$
\begin{array}{|l}
\hline
\Delta C\_Machine\_State \\
C\_Machine\_State \\
C\_Machine\_State' \\
next : \mathbb{N} \\
\delta Reg : Regs \pfun \mathbb{N} \\
\delta Lispstack : \mathbb{N} \pfun CL\_FORM \\
\ldots \\
\hline
C\_Code' = C\_Code \\
next = Reg(PC) + 1 \\
Reg' = Reg \oplus \{PC \mapsto next\} \oplus \delta Reg \\
Lispstack' = Lispstack \oplus \delta Lispstack \\
\ldots \\
\hline
\end{array}
$$

In contrast to the convention in [Spi92] the symbol Δ is used here not as a Z schema operator but as a normal symbol, which, used as prefix, hints for changes in state by convention.

To describe state transitions a predecessor and a successor state must be defined. The state transition will be defined by predicates about predecessor and successor states.

The local variable *next* stands for the value, the program counter will get in the normal case.

Both partial functions δReg and $\delta Lispstack$ describe changes in the registers and the lisp stack respectively. Changes on the C-stack are modelled in each individual instruction in another way.

The instruction `LOADFIXNUM` writes a small integer into the Lisp stack:

$$
\begin{array}{|l}
\hline
Loadfixnum \\
\Delta C\_Machine\_State \\
literal, position : \mathbb{N} \\
\hline
Op = LOADFIXNUM \\
literal = arg1 \\
position = arg2 \\
\delta Reg = \emptyset \\
C\_Stack' = C\_Stack \\
\delta Lispstack = \{position + Reg(BASE) \mapsto literal\} \\
\ldots \\
\hline
\end{array}
$$

Since δReg is defined to be empty, `LOADFIXNUM` changes the registers as specified in the normal case: the `PC` is incremented by one. The C-stack remains unchanged and the first argument of the instruction is written to the location determined by the 2nd argument relative to the start of the current stack frame.

The following schema specifies how to call a Lisp function with a known and fixed number of arguments:

$$
\begin{array}{|l}
\hline
\textit{Call} \\
\Delta C\_Machine\_State \\
target, framesize : \mathbb{N} \\
\hline
Op = CALL \\
target = arg1 \\
framesize = arg2 \\
\delta Reg = \{PC \mapsto target,\ BASE \mapsto Reg(BASE) + framesize\} \\
C\_Stack' = < Reg(PC), Reg(BASE) > {}^\frown C\_Stack \\
\delta Lispstack = \emptyset \\
\ldots \\
\hline
\end{array}
$$

The program counter is not modified as specified in the normal case, but is set to the start address of the called function. The register `BASE` is changed to point to the new stack frame. The old values of `PC` and `BASE` are saved on the C-stack. By this, Lisp data is never stored on the C-stack and the memory manager never needs to look for Lisp data there. The Lisp stack is not changed by the function call, because the calling function has already calculated arguments and stored them at appropriate positions by explicit operations.

After completely executing a function, its body is left and control is transfered to the callee:

$$
\begin{array}{|l}
\hline
\textit{Ret} \\
\Delta C\_Machine\_State \\
retaddr, oldbase : \mathbb{N} \\
\hline
Op = RET \\
\delta Reg = \{PC \mapsto retaddr + 1,\ BASE \mapsto oldbase\} \\
< retaddr, oldbase > {}^\frown C\_Stack' = C\_Stack \\
\delta Lispstack = \emptyset \\
\ldots \\
\hline
\end{array}
$$

The original value of `PC` and `BASE` which are put by `CALL` on the C-stack, are used to restore `PC` and `BASE`. The execution of the next instruction takes place at the address, which directly follows the `CALL` instruction in program memory. The Lisp stack remains unchanged.

6 Conclusion

C is established as a portable implementation language for operating systems and many application programs. To compile highlevel languages to C, highlevel language constructs must be simulated in C. Since branch targets in C are no first class values and the C runtime stack cannot be accessed explicitely, sometimes translations must be more complicated than necessary. However, the results reached today are already better than those produced by native code compilers with ad hoc code generators. In addition, C makes portable integration with standard software possible.

We can define the semantics of abstract C-machines in a precise way. This allows to systematically examine translation to C.

The idea to use C as target language will have feed back to C itself.[6] There is hope, that new C implementations will come into existance, which will better handle the requirements of automatically generated C-Code.

[6]The GNU-C-compiler, for example, already supports branch targets as C data.

References

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

[Bar89] J. F. Bartlet. Scheme→C: A portable Scheme-to-C compiler. Technical report, DEC Western Research Laboratory, 100 Hammilton Avenue, Palo Alto CA 94301, USA, January 1989.

[BMS81] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. HOPE, an experimental applicative language. internal report, Proccedings of the 1980 Lisp Conference, Stanford, CA CSR-62-80, University of Edinburgh, Department of Computer Science, 1981.

[Bow89] Jonathan Bowen. Z specification of the μtransputer instruction set. ProCoS Technical Report ProCos-PRG-001, Oxford University Computing Laboratory, Oxford, July 1989.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Pronciples of Programming Languages*, pages 297–302, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation.* Addison Wesley, Reading Mass., 1983.

[Jac82] Christian Jacobi. *Code Generation and the LILITH Architecture.* PhD thesis, ETH Zürich Nr.7195, Zürich, 1982.

[Joh84] J. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction*, pages 58–59, 1984.

[Lan63] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[PD82] S. Pemberton and M. C. Daniels. *Pascal Implementation: the P4 Compiler.* Ellis Horwood Ltd., Chichester, 1982.

[PJS89] S. L. Peyton Jones and J. Salkild. The spineless tagless G-machine. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, 1989.

[Ric71] M. Richards. The portability of the BCPL compiler. *Software Practice and Experience*, 1, 1971.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd edition, 1992.

[TAL91] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, 1991.

[War83] David S. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[Wir71] N. Wirth. The design of a Pascal compiler. *Software Practice and Experience*, 1:309–333, 1971.

[Wir81] N. Wirth. The personal computer Lilith. Technical report, ETH Zürich, 1981.