Compiling for the Apoo Virtual Machine

C. Amaral, R. Cruz, M. Florido, N. Moreira, R. Reis

DCC-FC & LIACC, Universidade do Porto, Porto, Portugal

Apoo: a virtual machine

Apoo[1] is an environment for programming in a simple assembly language suitable to teach the basic concepts of computer architecture, instructions set and operation without being obscured by the specific details of a real microprocessor.
The **Apoo virtual machine** is a virtual processor with a very simple architecture and instruction set that mimics almost all the essential features of a modern microprocessor.
The **Apoo Interface** is a graphical environment that monitors the state of the machine during the execution of a program and allows the writing/editing/execution of programs in assembly language.

(The Basic) Apoo Virtual Machine

Apoo has a set of general purpose registers, a data memory area, a program memory area, a system stack and a program counter register. Each register or memory cell can hold a 32-bits integer. Memory cells are created as needed by means of two different pseudo-instructions:

- the `mem` pseudo-instruction reserves an array of cells;
- the `const` pseudo-instruction reserves individual cells initializing them with the given values.

Each program memory cell will hold a whole instruction. The program counter, as usual, will contain the address of the next instruction to be executed. Finally, the system stack is used to implement subroutines and argument passing.

Apoo Instruction Set

Operation	Operand1	Operand2	Meaning
Data to Register Transfer			
`load`	`Mem`	`Ri`	Loads the contents of memory address `Mem` into register `Ri`;
`loadn`	`Num`	`Ri`	Loads number `Num` into register `Ri`;
`loadi`	`Ri`	`Rj`	Loads the contents of memory whose address is the contents of `Ri` into `Rj` (indirect load)
Data to Memory Transfer			
`store`	`Ri`	`Mem`	Stores the contents of `Ri` at memory address `Mem`;
`storer`	`Ri`	`Rj`	Stores the contents of `Ri` into `Rj`
`storei`	`Ri`	`Rj`	Stores the contents of `Ri` into memory cell whose address is the contents of `Rj`
Data to the System Stack Transfer			
`push`	`Ri`		Pushes the contents of `Ri` into the top of the stack
`pop`	`Ri`		Pops the element from the top of the stack into `Ri`
Two Operand Arithmetic			
`add`	`Ri`	`Rj`	`Rj=Ri+Rj`)
`sub`	`Ri`	`Rj`	(`Rj=Ri-Rj`)
`mul`	`Ri`	`Rj`	(`Rj=Ri*Rj`)
`div`	`Ri`	`Rj`	(`Rj=Ri/Rj`)
`mod`	`Ri`	`Rj`	(`Rj=Ri%Rj`)
One Operand Arithmetic			
`zero`	`Ri`		Stores 0 in `Ri` (`Ri=0`)
`inc`	`Ri`		Increments by 1 the contents of `Ri` (`Ri=Ri+1`)
`dec`	`Ri`		Decrements by 1 the contents of `Ri` (`Ri=Ri-1`)
Control Transfer			
`jump`	`Addr`		Jumps to instruction at address `Addr`;
`jzero`	`Ri`	`Addr`	Jumps to instruction at address Addr, if the contents of `Ri` is zero;
`jpos`	`Ri`	`Addr`	Jumps to instruction at address `Addr`, if the contents of `Ri` is positive;
`jneg`	`Ri`	`Addr`	Jumps to instruction at address `Addr`, if the contents of `Ri` is negative;
`jsr`	`Addr`		Pushes the contents of `PC` into the stack and jumps to instruction at address `Addr`
`rtn`			Pops an address from the top of the stack into the `PC`
`halt`			Stops execution;

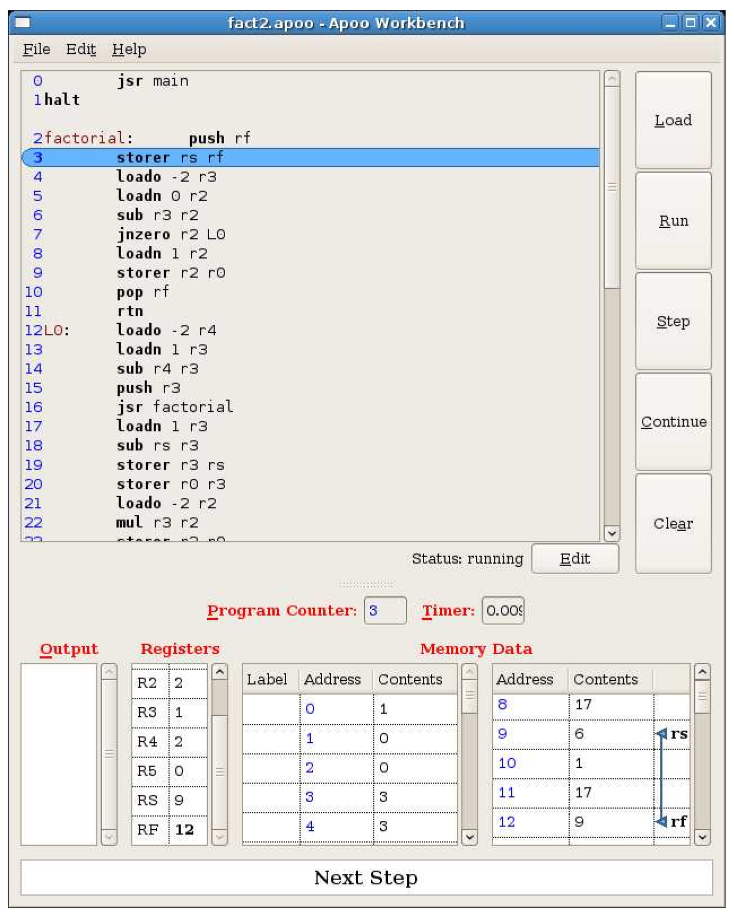
Manipulation of activation records

In order to allow the implementation of functions with local information, the **Apoo** memory model was modified to allow the manipulation of activation records. The size of the RAM is now predefined and divided into two areas: static memory and system stack. The static memory, begins at address 0 and it is allocated when an **Apoo** program is loaded (corresponding to the memory reserved using `const` and `mem`). The system stack occupies the rest of the RAM (growing for higher addresses). There are two programmable registers to address the system stack: stack register (`rs`) and frame register (`rf`). The stack register `rs` contains the address of the last stack memory cell (or -1 if no static memory is allocated). The instructions `jsr`, `rtn`, `push` and `pop` manipulates the stack in the usual way. The frame register can be used for the implementation of local information. It is also used in two special instructions:

`storeo` $\mathtt{R_i}$ `Num` stores the contents of register R_i at memory address $(rf) + Num$.

`loado` `Num` $\mathtt{R_i}$ loads the contents of memory address $(rf) + Num$ into register R_i.

Apoo Interface



The program in execution is the compiled version of the following `C` program for the factorial:

```
int factorial(int n) {
  if (n==0) return 1;
  return factorial(n-1)*n;}
int main(){
  int k, fact;
  scanf(k); fact = factorial(k);
  printf (fact);}}
```

[1] www.ncc.up.pt/apoo

Compilation

The compiler

Here we describe the main course work in teaching Compilers at the Faculty of Science of the University of Porto (FCUP). This work consists of a compiler of a subset of the mainstream C programming language. The structure is a typical one. It consists of the usual phases in the compilation process: lexical analysis, syntax analysis, semantic analysis, machine independent code generation, storage allocation and code generation. However the techniques and tools of our approach have some novel features with clear advantages with respect to more traditional frameworks:

1. **Code generation**: we generate **Apoo** code. This enables the student to focus on the relevant part of the code generation phase, and not on the annoying and tedious specific characteristics of a real machine code.
2. **Haskell**: we used the Haskell programming language for the implementation of our compiler. Due to the high declarative nature of Haskell the code becomes much more readable.
3. **Tools**: Haskell has all the usual compilation tools. We used Alex for the lexical analysis, and Happy for the parser. This tools relieve the student of many of the tedious and error-prone aspects of producing compilers.
4. **Minimal compiler size and ease of maintenance**: a declarative approach enables the production of small compilers and at the same time with code which is ease to read and maintain.

Happy Parsing

Here is fragment of a C grammar.
For comands:

```
comm ::                     {Comando}
comm : atrib                 {CmdA $1}
| if_cmd                     {CmdC $1}
| proc                       {Call $1}
| cmd_block                  {CmdB $1}
| pre_defs_io                {$1}
| return_line                {$1}

return_line ::               {Comando}
return_line : RETURN ';'{Return Nothing}
| RETURN expar ';'           {Return (Just $2)}

atrib ::                {Atribuicao}
atrib : ID '=' expar ';'  {Atrib $1 $3}
if_cmd ::                 {Condicao}
if_cmd : IF '(' expr_lg ')' comm
               ELSE comm   {IfElse $3 $5 $7}
| IF '(' explg ')' comm    {If $3 $5}
```

For expressions:

```
expar ::                     {ExpAr}
expar : INTEIRO              {Int (snd $1)}
| IDENTIF                    {Var $1}
| f_call                     {Fnc $1}
| expar '+' expar            {Add $1 $3}
| expar '-' expar            {Sub $1 $3}
| expar '/' expar            {Div $1 $3}
| expar '*' expar            {Mul $1 $3}
| expar '%' expar            {Mod $1 $3}
| '-' expar                  {Sim $2}
| '(' expar ')'              {$2}

f_call ::                    {Chamada}
f_call : ID '('explst')' {($1, $3)}

explst ::                    {[ExpAr]}
explst : expar               {[$1]}
| explst ',' expar           {$3:$1}
```

Abstract Syntax for a subset of `C`

```
data Comando = CmdA Atribuicao
             | CmdC Condicao
             | Call Chamada
             | CmdB BlocoComandos
             | Scan Name
             | Print ExprAr
             | Return (Maybe ExpAr)
          deriving(Eq, Show)

data Atribuicao = Atrib Name ExpAr
             deriving(Eq, Show)

data Condicao = If ExprLg Comando
    | IfElse ExprLg Comando Comando
    deriving(Eq, Show)

data ExpAr = Add ExpAr ExpAr
           | Sub ExpAr ExpAr
           | Mul ExpAr ExpAr
           | Div ExpAr ExpAr
           | Mod ExpAr ExpAr
           | Sim ExpAr
           | Int Int
           | Var Name
           | Fnc Chamada
           deriving(Eq, Show)
```

Compiler top level

```
main :: IO ()
main = (getContents >>=
  (interCodeGen.parser.scanner))
  >>= (myPrint.theApGen)
```

Intermediate code generation

```
genEArICode :: SymTb -> ExpAr -> Int -> IO(Tmp)
genEArICode st (Int c) l = return (CONST c)
genEArICode st (Var n) l = getOffset st n l >>=
               \offset -> return (MEM (VAR offset))
genEArICode st (Add e1 e2) l = mkBinO st PLUS e1 e2 l
genEArICode st (Sub e1 e2) l = mkBinO st MINUS e1 e2 l
genEArICode st (Mul e1 e2) l = mkBinO st MUL e1 e2 l
genEArICode st (Div e1 e2) l = mkBinO st DIV e1 e2 l
genEArICode st (Mod e1 e2) l = mkBinO st MOD e1 e2 l
genEArICode st (Sim e) l = mkBinO st MINUS (Int 0) e l

mkBinO::SymTb -> BO -> ExpAr -> ExpAr -> Int -> IO(Tmp)
mkBinO st op e1 e2 l =
    genEArICode st e1 l >>=
       \t1 -> genExpArICode st e2 l >>=
       \t2 ->  return (BINOP op t1 t2)
```

Code generation for functions in **Apoo**

```
theApGen :: Atree -> [Instruction]
theApGen at = [ApJSR "main",ApHALT] ++ (apooGen at "r1")
apooGen :: Atree -> LastReg -> [Instruction]
apooGen [] _ = []
apooGen (((lbl,nargs),stmts):ats) lt =
 (ApBlankLine:(ApLbl lbl):fBody) ++ (apooGen ats lt)
    where
    insts = apooStmtGen stmts lt
    fBody = fPrologue ++ insts ++ fEpilogue nargs lt

fPrologue = [ApPush "rf",ApStorer "rs" "rf"]
fEpilogue n lt = [ApLoadn n lt,ApSub "rs" lt,
               ApStorer lt "rs",ApPop "rf",ApRTN]
```