Chapter 1:  
The Register C-Machine

The Register C-Machine (R-CMa)

We generate Code for the Register C-Machine.
The Register C-Machine is a virtual machine (VM).
- there exists no processor that can execute its instructions
- ... but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no `double`, `float`, `char`, `short` or `long` types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers
Virtual Machines

A virtual machine has the following ingredients:
- any virtual machine provides a set of instructions
- instructions are executed on virtual hardware
- the virtual hardware is a collection of data structures that is accessed and modified by the VM instructions
- ... and also by other components of the run-time system, namely functions that go beyond the instruction semantics
- the interpreter is part of the run-time system

Executing a Program

- the machine loads an instruction from [PC] into the instruction register IR in order to execute it
- before evaluating the instruction, the PC is incremented by one

```c
while (true) {
    IR = C[PC]; PC++;
    execute {IR};
}
```

- note: the PC must be incremented before the execution, since an instruction may modify the PC
- the loop is exited by evaluating a halt instruction that returns directly to the operating system

Components of a Virtual Machine

Consider Java as an example:

- C: code
- S: stack

A virtual machine such as the Dalvik VM has the following structure:
- S: the data store -- a memory region in which cells can be stored in LIFO order -> stack
- SP (stack pointer) pointer to the last used cell in S
- beyond S follows the memory containing the heap

Chapter 2:
Generating Code for the Register C-Machine
Principles of the R-CMa

The R-CMa is composed of a stack, heap and a code segment, just like the JVM; it additionally has register sets:
- local registers are $R_1, R_2, \ldots, R_m, \ldots$
- global registers are $R_0, R_{-1}, \ldots, R_{-n}, \ldots$

C

0 1

S

0

$R_{loc}$

$R_1$ $R_0$

$R_{glob}$

$R_0$ $R_{-4}$

PC

SP

The Register Sets of the R-CMa

The two register sets have the following purpose:
- the local registers $R_i$
  - save temporary results
  - store the contents of local variables of a function
  - can efficiently be stored and restored from the stack

Translation of Expressions

Using variables stored in registers; loading constants:

<table>
<thead>
<tr>
<th>instruction</th>
<th>semantics</th>
<th>intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $e_1$ (R_i) (R_j)</td>
<td>(R_i = e_1)</td>
<td>load constant (R_j) to (R_i)</td>
</tr>
<tr>
<td>move (R_i) (R_j)</td>
<td>(R_i = R_j)</td>
<td></td>
</tr>
</tbody>
</table>

Let \(\text{op} = \{\text{add}, \text{sub}, \text{div}, \text{mul}, \text{le}, \text{gt}, \text{ge}, \text{eq}, \text{neq}, \text{and}, \text{or}\}\).

The R-CMa provides an instruction for each operator \(\text{op}\).

\[
\text{op} \ R_i \ R_j \ R_k
\]

where $R_i$ is the target register, $R_j$ the first and $R_k$ the second argument.

Correspondingly, we generate code as follows:

\[
\begin{align*}
\text{code}_{k+1} & \ e_1 \ \text{op} \ e_2 \ \rho \\
\text{code}_{k+2} & \ e_3 \ \rho \\
\text{code}_{k+3} & \ \text{op} \ R_i \ R_j \ R_{k+3}
\end{align*}
\]
Managing Temporary Registers

Observe that temporary registers are re-used: translate $3 \times 4 + 3 \times 4$ with $i = 4$.

$$
code_h^{3 \times 4 + 3 \times 4} \rho = code_h^{3 \times 4} \rho
$$

where

$$
code_h^{3 \times 4} \rho = load R, 3
load R_{i-1}, 4
mul R_i, R, R_{i+1}
$$

we obtain

$$
code_h^{3 \times 4 + 3 \times 4} \rho =
$$

Semantics of Operators

The operators have the following semantics:

- **add**: $R_i = R_j + R_k$
- **sub**: $R_i = R_j - R_k$
- **div**: $R_i = R_j / R_k$
- **mul**: $R_i = R_j \times R_k$
- **mod**: $R_i = \text{signum}(R_j) \cdot R_k$

where

$$
\text{signum}(R_j) = \begin{cases} 1 & \text{if } j < 0 \\ 0 & \text{otherwise} \end{cases}
$$

- **le**: $R_i = 1$ if $R_j < R_k$ and $0$ otherwise
- **ge**: $R_i = 1$ if $R_j \geq R_k$ and $0$ otherwise
- **eq**: $R_i = 1$ if $R_j = R_k$ and $0$ otherwise
- **neq**: $R_i = 1$ if $R_j \neq R_k$ and $0$ otherwise
- **and**: $R_i = R_j \& R_k$ bit-wise and
- **or**: $R_i = R_j | R_k$ bit-wise or

Translation of Unary Operators

Unary operators $\text{op} = \{ \text{neg}, \text{not} \}$ take only two registers:

$$
code_h^{\text{op}} \rho = code_h^{\text{op}} \rho
\begin{array}{l}
\text{op} R_i, R_j
\end{array}
$$

Applying Translation Schema for Expressions

Suppose the following function $f$ is given:

```c
void f(void) {
    int x, y, z;
    x = y + z;
}
```

- Let $\rho = \{ x \mapsto 1, y \mapsto 2, z \mapsto 3 \}$ be the address environment.
- Let $R_i$ be the first free register, that is, $i = 4$.

$$
code^4 x = y + z + 3 \rho = code^4 y + z + 3 \rho
\begin{array}{l}
\text{move} R_i, R_i
\end{array}
$$
Chapter 3: Statements and Control Structures

Simple Conditional

We first consider \( s = i \), and present a translation without basic blocks.

Idea:
- emit the code of \( c \) and \( ss \) in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

\[
\text{code}^s \ s \ p = \begin{cases}
\text{code}^c \ c \ p & \text{if } c \text{ is true} \\
\text{jumpz } R, A & \text{if } c \text{ is false}
\end{cases}
\]

Translation of Statement Sequences

The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

\[
\begin{align*}
\text{code}^s (s \ ss) \ p &= \text{code}^s \ s \ p \\
\text{code}^s \ ss \ p &= \text{code}^s \ s \ p \\
\text{code}^s \ \epsilon \ p &= \text{# empty sequence of instructions}
\end{align*}
\]

Note here: \( s \) is a statement, \( ss \) is a sequence of statements

General Conditional

Translation of \( \text{if} (c) \ tt \text{ else } ce \)

\[
\text{code}^i \text{f}(c) \ tt \text{ else } ce \ p = \begin{cases}
\text{code}^c \ c \ p & \text{if } c \text{ is true} \\
\text{jumpz } R, A & \text{if } c \text{ is false}
\end{cases}
\]

\[
\begin{align*}
\text{code}^c \ c \ p &= \text{code for } c \\
\text{jumpz } R, A & = \text{code for } tt \\
\text{jump } B & = \text{code for } ce \\
\end{align*}
\]
Example for if-statement

Let \( \rho = \{ x \rightarrow 4, y \rightarrow 7 \} \) and let \( s \) be the statement

\[
\text{if } x > y \text{ then } x := x - 1; \quad \text{// (i) } \ast /
\]
\[
\text{else } y := y - x; \quad \text{// (ii) } \ast /
\]
\[
y := y - x; \quad \text{// (iii) } \ast /
\]

Then \( \text{code}^s \rho \) yields:

for-Loops

The for-loop \( s = \textbf{for } (e_1; e_2; e_3) \textbf{ s}' \) is equivalent to the statement sequence \( e_1; \textbf{while } (e_2) \{ s' \textbf{ e}_3; \} \) as long as \( s' \) does not contain a continue statement.

Thus, we translate:

\[
\text{code}^s \textbf{for} (e_1; e_2; e_3) \textbf{ s} \rho = \text{code}^s_1 e_1 \rho \quad A : \quad \text{code}^s_2 e_2 \rho \quad \text{jump } R_1, B \\
\text{code}^s s \rho \quad \text{code}^s_3 e_3 \rho \quad \text{jump } A
\]

Iterating Statements

We only consider the loop \( s = \textbf{while } (e) \textbf{ s}' \). For this statement we define:

\[
\text{code}^s \textbf{while} (e) \textbf{ s} \rho = A : \quad \text{code}^s_1 e \rho \\
\text{jump } R_1, B \\
\text{code}^s s \rho \\
\text{code}^s \textbf{ for } e \\
\text{code}^s \textbf{ for } s'
\]

Consecutive Alternatives

Let \textbf{switch } be given with \( k \) consecutive \textbf{case} alternatives:

\[
\text{switch } (e) \{
\text{case } 0 : s_0; \textbf{ break}; \\
\vdots \\
\text{case } k-1 : s_{k-1}; \textbf{ break}; \\
\text{default : } s_k; \textbf{ break}; \\
\}
\]
Translation of the \textit{check}^1 Macro

The macro \textit{check}^1 l u B checks if l ≤ R_i < u. Let k = u - l.
- If l ≤ R_i < u it jumps to B + R_i - l
- If R_i < l or R_i ≥ u it jumps to A_k

\[
B : \quad \text{jump } A_0 \\
C : \quad \text{jump } A_k
\]

improvements for Jump Tables

This translation is only suitable for \textit{certain switch-statement}.
- In case the table starts with \( i \) instead of \( u \) we don't need to subtract \( \epsilon \) before we use it as index.
- If the value of \( \epsilon \) is \textit{guaranteed} to be in the interval \([l, u]\), we can omit \textit{check}.

General translation of switch-Statements

In general, the values of the various cases may be far apart:
- Generate an \textit{if}-ladder, that is, a sequence of \textit{if}-statements.
- For \( n \) cases, an \textit{if}-cascade (tree of conditionals) can be generated \( \sim O(\log n) \) tests.
- If the sequence of numbers has small gaps (≤ 3), a jump table may be smaller and faster.
- One could generate several jump tables, one for each set of consecutive cases.
- An \textit{if} cascade can be re-arranged by using information from \textit{profiling}, so that paths executed more frequently require fewer tests.
Chapter 4:
Functions

Memory Management in Functions

```
int fac(int x) {
    if (x<=0) return 1;
    else return x*fac(x-1);
}

int main(void) {
    int n;
    n = fac(2) + fac(1);
    printf("%d", n);
}
```

At run-time several instances may be active, that is, the function has been called but has not yet returned.
The recursion tree in the example:

```
main
   fac
   fac
   printf
   fac
   fac
```

Memory Management in Function Variables

The formal parameters and the local variables of the various instances of a function must be kept separate

Idea for implementing functions:

- set up a region of memory each time it is called
- in sequential programs this memory region can be allocated on the stack
- thus, each instance of a function has its own region on the stack
- these regions are called stack frames

Ingredients of a Function

The definition of a function consists of
- a name with which it can be called;
- a specification of its formal parameters;
- possibly a result type;
- a sequence of statements.

In C we have:

```
code[\_f \rho \_f] = \text{load } R_{\_f} \_f \text{ with } \_f \text{ starting address of } f
```

Observe:
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later
Organization of a Stack Frame

- stack representation: grows upwards
- SP points to the last used stack cell

local memory
callee
organizational
cells

Split of Obligations

**Definition**

Let $f$ be the current function that calls a function $g$.
- $f$ is dubbed **caller**
- $g$ is dubbed **callee**

The code for managing function calls has to be split between caller and callee. This split cannot be done arbitrarily since some information is only known in that caller or only in the callee.

**Observation:**

The space requirement for parameters is only know by the caller:

Example: `printf`

Principle of Function Call and Return

**actions taken on entering $g$:**

1. compute the start address of $g$
2. compute actual parameters in globals
3. backup of caller-save registers
4. backup of FP
5. set the new FP
6. back up of PC and jump to the beginning of $g$
7. copy actual params to locals

**actions taken on leaving $g$:**

1. compute the result into $R_0$
2. restore FP, SP
3. return to the call site in $f$, that is, restore PC
4. restore the caller-save registers

Managing Registers during Function Calls

The two register sets (global and local) are used as follows:
- automatic variables live in **local registers** $R_i$
- intermediate results also live in **local registers** $R_i$
- parameters live in **global registers** $R_i$ (with $i < 0$)
- global variables:
Translation of Function Calls
A function call $p(e_1, \ldots, e_n)$ is translated as follows:

\[
\begin{align*}
\text{code}_1^\rho \cdot p & = \text{code}_n^\rho \cdot p \\
& \quad \vdots \\
& \quad \text{code}_1^\rho \cdot e_n \\
\text{move} \; R_1, R_{n+1} \\
& \quad \vdots \\
\text{move} \; R_n, R_{2n+1} \\
\text{save LOC} \; R_1, R_{n-1} \\
\text{mark} \\
\text{call} \; R_i \\
\text{restore LOC} \; R_1, R_{n-1} \\
\text{move} \; R_n, R_0
\end{align*}
\]

Return from a Function
The instruction return relinquishes control of the current stack frame, that is, it restores PC and FP.

\[\begin{align*}
\text{PC} = \text{S}[FP]; \\
\text{SP} = \text{FP} - 2; \\
\text{FP} = \text{S}[SP + 1];
\end{align*}\]

Rescuing the FP
The instruction mark allocates stack space for the return value and the organizational cells and backs up FP.

\[\begin{align*}
\text{FP} & \quad \text{mark} \\
\text{FP} & \quad \text{SP} = \text{SP} + 1;
\end{align*}\]

Translation of Whole Programs
A program $P = F_1, \ldots, F_n$ must have a single main function.

\[\begin{align*}
\text{code}_1^\rho \cdot P & = \text{load} \; R_{\text{main}} \\
& \quad \text{mark} \\
& \quad \text{call} \; R_1 \\
& \quad \text{halt} \\
\text{halt} & \quad \vdots \\
\text{halt} & \quad \text{for} \; F_1, \ldots, F_n
\end{align*}\]
Translation of the \texttt{fac}-function

Consider:

\begin{verbatim}
int fac(int x) {
  if (x==0) then
    return 1;
  else
    return x*fac(x-1);
}
\end{verbatim}

\begin{verbatim}
_x: move R3 R1
   move R3 R1
   load R3 0
   leq R2 R3 R3
   jumpz R2 A
   load R2 1
   move R0 R2
   return
   jump _B

   code is dead $\Box$

A: move R3 R1
   move R3 R1
   load R1 1
   sub R3 R3 R4
   move R1-1 R3
   load R3 fac
   saveloc R1 R2
   mark
   call R3
   restoreloc R1 R2
   move R2 R0
   mul R2 R2 R3
   move R0 R2
   return

\end{verbatim}

\begin{verbatim}
B: x*fac(x-1)
   x-1
   fac(x-1)
\end{verbatim}