Script **generated by TTT**

Title: Petter: Compilerbau (27.07.2017)

Date: Thu Jul 27 14:15:17 CEST 2017

Duration: 97:31 min

Pages: 41

Topic:

Code Synthesis

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

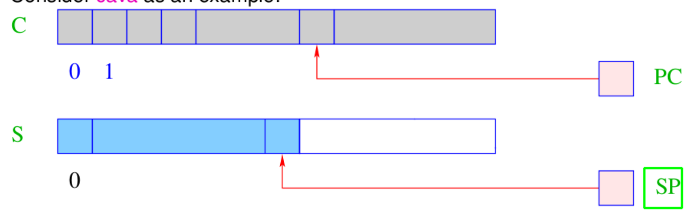
Virtual Machines

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

Components of a Virtual Machine

Consider Java as an example:



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

Executing a Program

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

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

- node: 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

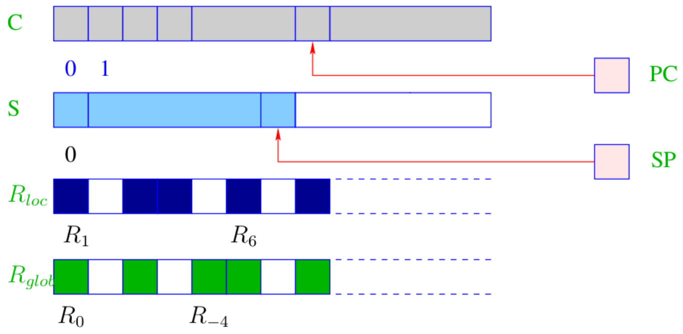
Chapter 2:

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 $R_1, R_2, \ldots R_i, \ldots$
- *global* register are $R_0, R_{-1}, \ldots R_j, \ldots$



The Register Sets of the R-CMa

The two register sets have the following purpose:

1. 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 Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
loadc R_i c	$R_i = c$	load constant
move R_i R_j	$R_i = R_j$	copy R_j to R_i

Translation of Expressions

Let op $= \{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or\}$.
The R-CMa provides an instruction for each operator 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:

$$\begin{aligned} \mathrm{code}_R^i\ e_1 \text{ op } e_2\ \rho \quad = \quad & \mathrm{code}_R^i\ e_1\ \rho \\ & \mathrm{code}_R^{i+1}\ e_2\ \rho \\ & \text{op } R_i\ R_i\ R_{i+1} \end{aligned}$$

Managing Temporary Registers

Observe that temporary registers are re-used: translate `3*4+3*4` with $t = 4$:

$$\begin{array}{lcl} \text{code}^4_R\ \texttt{3*4+3*4}\ \rho & = & \text{code}^4_R\ \texttt{3*4}\ \rho \\ & & \text{code}^5_R\ \texttt{3*4}\ \rho \\ & & \text{add}\ R_4\ R_4\ R_5 \end{array}$$

where

$$\begin{array}{lcl} \text{code}^i_R\ \texttt{3*4}\ \rho & = & \text{loadc}\ R_i\ 3 \\ & & \text{loadc}\ R_{i+1}\ 4 \\ & & \text{mul}\ R_i\ R_i\ R_{i+1} \end{array}$$

we obtain

$$\text{code}^4_R\ \texttt{3*4+3*4}\ \rho \quad =$$

Semantics of Operators

The operators have the following semantics:

add R_i R_j R_k	$R_i = R_j + R_k$
sub R_i R_j R_k	$R_i = R_j - R_k$
div R_i R_j R_k	$R_i = R_j / R_k$
mul R_i R_j R_k	$R_i = R_j * R_k$
mod R_i R_j R_k	$R_i = signum(R_k) \cdot k$ with $\lvert R_j \rvert = n \cdot \lvert R_k \rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k \rvert$
le R_i R_j R_k	$R_i =$ if $R_j < R_k$ then 1 else 0
gr R_i R_j R_k	$R_i =$ if $R_j > R_k$ then 1 else 0
eq R_i R_j R_k	$R_i =$ if $R_j = R_k$ then 1 else 0
leq R_i R_j R_k	$R_i =$ if $R_j \leq R_k$ then 1 else 0
geq R_i R_j R_k	$R_i =$ if $R_j \geq R_k$ then 1 else 0
and R_i R_j R_k	$R_i = R_j \ \& \ R_k$ // bit-wise and
or R_i R_j R_k	$R_i = R_j \mid R_k$ // bit-wise or

Translation of Unary Operators

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

$$\begin{array}{lcl} \text{code}^i_R\ \text{op}\ e\ \rho & = & \text{code}^i_R\ e\ \rho \\ & & \text{op}\ R_i\ R_i \end{array}$$

Applying Translation Schema for Expressions

Suppose the following function is given:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ be the address environment.
- Let R_4 be the first free register, that is, $i = 4$.

$$\begin{array}{lcl} \text{code}^4\ \texttt{x=y+z*3}\ \rho & = & \text{code}^4_R\ \texttt{y+z*3}\ \rho \\ & & \text{move}\ R_1\ R_4 \end{array}$$

Chapter 3:

Statements and Control Structures

Translation of Statement Sequences

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

$$
\begin{array}{lcl}
\text{code}^i\ (s\ ss)\ \rho & = & \text{code}^i\ s\ \rho \\
 & & \text{code}^i\ ss\ \rho \\
\text{code}^i\ \varepsilon\ \rho & = & /\!/\quad \textit{empty sequence of instructions}
\end{array}
$$

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

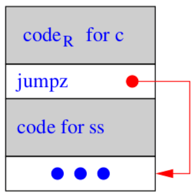
Simple Conditional

We first consider $s \equiv$ **if** $(\ c\ )\ ss$.
...and present a translation without basic blocks.

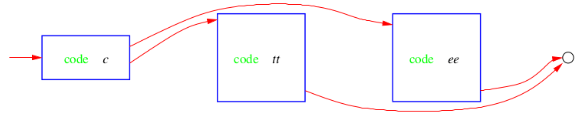
Idea:

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

$$
\begin{array}{rcl}
\text{code}^i\ s\ \rho & = & \text{code}^i_{\text{R}}\ c\ \rho \\
 & & \text{jumpz}\ R_i\ A \\
 & & \text{code}^i\ ss\ \rho \\
 & A: & \dots
\end{array}
$$

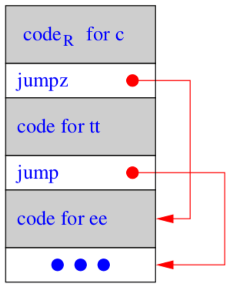


General Conditional



Translation of **if** $(\ c\ )\ tt$ **else** ee.

$$
\begin{array}{rcl}
\text{code}^i\ \texttt{if}(c)\ tt\ \texttt{else}\ ee\ \rho & = & \\
 & & \text{code}^i_{\text{R}}\ c\ \rho \\
 & & \text{jumpz}\ R_i\ A \\
 & & \text{code}^i\ tt\ \rho \\
 & & \text{jump}\ B \\
 & A: & \text{code}^i\ ee\ \rho \\
 & B: &
\end{array}
$$



Example for if-statement

Let $\rho = \{x \mapsto 4, y \mapsto 7\}$ and let s be the statement

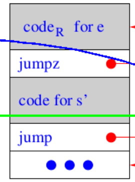
```
if (x>y) {          /* (i)   */
  x = x - y;        /* (ii)  */
} else {
  y = y - x;        /* (iii) */
}
```

Then $\mathrm{code}^i\ s\ \rho$ yields:

Iterating Statements

We only consider the loop $s \equiv$ **while** $(e)\ s'$. For this statement we define:

$$\begin{aligned} \mathrm{code}^i\ \texttt{while}(e)\ s\ \rho \quad = A: \quad & \mathrm{code}^i_{\mathrm{R}}\ e\ \rho \\ & \mathrm{jumpz}\ R_i\ B \\ & \mathrm{code}^i\ s\ \rho \\ & \mathrm{jump}\ A \\ B: \quad & \end{aligned}$$



for-Loops

The **for**-loop $s \equiv$ **for** $(e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;$ **while** $(e_2)\ \{s'\ e_3;\}$ – as long as s' does not contain a **continue** statement.

Thus, we translate:

$$\begin{aligned} \mathrm{code}^i\ \texttt{for}(e_1; e_2; e_3)\ s\ \rho \quad = \quad & \mathrm{code}^i_{\mathrm{R}}\ e_1\ \rho \\ A: \quad & \mathrm{code}^i_{\mathrm{R}}\ e_2\ \rho \\ & \mathrm{jumpz}\ R_i\ B \\ & \mathrm{code}^i\ s\ \rho \\ & \mathrm{code}^i_{\mathrm{R}}\ e_3\ \rho \\ & \mathrm{jump}\ A \\ B: \quad & \end{aligned}$$

Consecutive Alternatives

Let **switch** s be given with k consecutive **case** alternatives:

$$\begin{aligned} & \textbf{switch}\ (e)\ \{ \\ & \quad \textbf{case}\ 0\text{:}\ s_0\text{;}\ \textbf{break;} \\ & \quad \vdots \\ & \quad \textbf{case}\ k-1\text{:}\ s_{k-1}\text{;}\ \textbf{break;} \\ & \quad \textbf{default:}\ s_k\text{;}\ \textbf{break;} \\ & \} \end{aligned}$$

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ checks if $l \leq R_i < u$. Let $k = u - l$.

- if $l \leq R_i < u$ it jumps to $B + R_i - l$
- if $R_i < l$ or $R_i \geq u$ it jumps to A_k

$$
\begin{array}{rl}
B: & \text{jump } A_0 \\
\vdots & \quad \vdots \\
 & \text{jump } A_k \\
C: &
\end{array}
$$

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ checks if $l \leq R_i < u$. Let $k = u - l$.

- if $l \leq R_i < u$ it jumps to $B + R_i - l$
- if $R_i < l$ or $R_i \geq u$ it jumps to A_k

we define:

$$
\begin{array}{rrl}
check^i\ l\ u\ B & = & \text{loadc } R_{i+1}\ l \\
 & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz } R_{i+2}\ E \\
 & & \text{sub } R_i\ R_i\ R_{i+1} \\
 & & \text{loadc } R_{i+1}\ u \\
 & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz } R_{i+2}\ D \\
 & E: & \text{loadc } R_i\ u - l \\
 & D: & \text{jumpi } R_i\ B
\end{array}
\qquad
\begin{array}{rl}
B: & \text{jump } A_0 \\
\vdots & \quad \vdots \\
 & \text{jump } A_k \\
C: &
\end{array}
$$

Improvements for Jump Tables

This translation is only suitable for *certain* **`switch`**-statement.

- In case the table starts with 0 instead of u we don't need to subtract it from e before we use it as index
- if the value of e is guaranteed to be in the interval $[l, u]$, we can omit $check$

General translation of switch-Statements

In general, the values of the various cases may be far apart:

- generate an **`if`**-ladder, that is, a sequence of **`if`**-statements
- for n cases, an **`if`**-cascade (tree of conditionals) can be generated $\leadsto O(\log n)$ 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 sets of consecutive cases
- an **`if`** cascade can be re-arranged by using information from *profiling*, so that paths executed more frequently require fewer tests

Chapter 4:
Functions

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:

$$\text{code}^i_{\mathrm{R}}\ f\ \rho \quad = \quad \text{loadc}\ R_i\ \_f \qquad \text{with } \_f \text{ starting address of } f$$

Observe:

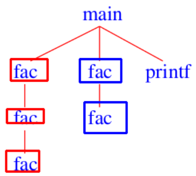
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later

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:



Memory Management in Function Variables

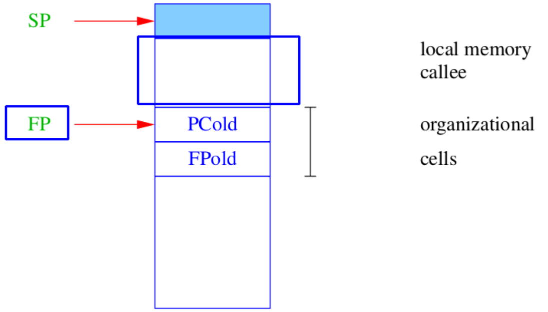
The formal parameters and the local variables of the various instances of a function must be kept separate

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

Organization of a Stack Frame

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



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

} saveloc
} mark
} call
} ... } is in g
} are in f

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

} return } are in g
} restoreloc } is in f

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 $i \leq 0$)
- global variables:

Translation of Function Calls

A function call $g(e_1, \dots e_n)$ is translated as follows:

$$
\begin{array}{lcl}
\text{code}^i_{\text{R}}\ g(e_1, \dots e_n)\ \rho & = & \text{code}^i_{\text{R}}\ g\ \rho \\
 & & \text{code}^{i+1}_{\text{R}}\ e_1\ \rho \\
 & & \vdots \\
 & & \text{code}^{i+n}_{\text{R}}\ e_n\ \rho \\
 & & \text{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \text{move}\ R_{-n}\ R_{i+n} \\
 & & \text{saveloc}\ R_1\ R_{i-1} \\
 & & \text{mark} \\
 & & \text{call}\ R_i \\
 & & \text{restoreloc}\ R_1\ R_{i-1} \\
 & & \text{move}\ R_i\ R_0
\end{array}
$$

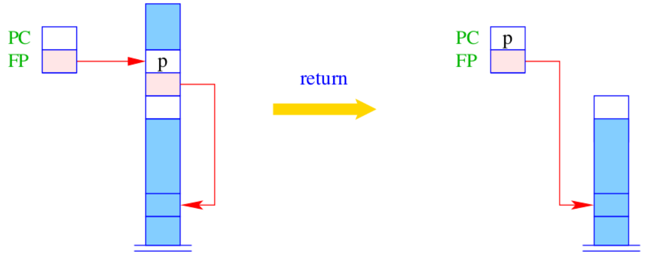
Rescuing the FP

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

S[SP+1] = FP;
SP = SP + 1;

Return from a Function

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



PC = S[FP];
SP = FP-2;
FP = S[SP+1];

Translation of Whole Programs

A program $P = F_1; \dots F_n$ must have a single `main` function.

$$
\begin{array}{lcl}
\text{code}^1\ P\ \rho & = & \text{loadc}\ R_1\ \_\text{main} \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \text{halt} \\
\_f_1: & & \text{code}^1\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^1\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Translation of the `fac`-function

Consider:

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

_fac:	move R_1 R_{-1}	save param.
$i = 2$	move R_2 R_1	`if (x<=0)`
	loadc R_3 0	
	leq R_2 R_2 R_3	
	jumpz R_2 $\_A$	to else
	loadc R_2 1	`return 1`
	move R_0 R_2	
	return	
	jump $\_B$	code is dead

_A:	move R_2 R_1	`x*fac(x-1)`
$i = 3$	move R_3 R_1	`x-1`
$i = 4$	loadc R_4 1	
	sub R_3 R_3 R_4	
$i = 3$	move R_{-1} R_3	`fac(x-1)`
	loadc R_3 $\_fac$	
	saveloc R_1 R_2	
	mark	
	call R_3	
	restoreloc R_1 R_2	
	move R_3 R_0	
	mul R_2 R_2 R_3	
	move R_0 R_2	`return x*...`
	return	
_B:	return	

Ende der Präsentation. Klicken Sie zum Schließen.