Introduction to Theoretical Computer Science

Notes

The purpose of these notes is to provide a concise and detailed reference for all definitions, concepts and theorems in the course. They largely repeat the slides, adding details where the slides omit them. They do not repeat examples that are on the slides. They are not a substitute for reading around, but in principle they contain everything you should know by the end of the course.

Part I: Computability

1 Register Machines

A register machine is a very simple mathematical model of a computer, conceptually similar to most actual computer architectures.

Definition 1 A register machine $M$ comprises
- a fixed set $R$ of $m$ registers $R_0, \ldots, R_{m-1}$, each of which is a natural number; we write $R_i$ both for the register and for its current value.
- A fixed program of length $n$ that is a sequence $P = I_0, I_1, \ldots, I_{n-1}$ of instructions, where
  - each instruction is one of:
    - $\text{INC}(i)$ where $0 \leq i < m$: add 1 to $R_i$, or
    - $\text{DEC}(i, j)$ where $0 \leq i < m$ and $0 \leq j \leq n$: if $R_i = 0$ then goto $I_j$, else subtract 1 from $R_i$. The pseudo-instruction $I_n$ represents a halting state.

The machine runs its program by executing the instructions from the start in order, except where a $\text{DEC}$ instruction causes a jump. We formalize this:

Definition 2 A state of a register machine $M$ as above is a tuple $(R_0, \ldots, R_{m-1}, C)$ where $R_i \in \mathbb{N}$ is the value of register $i$, and $0 \leq C \leq n$ is the program counter.

The next state function is a partial function from states to states, defined as follows:
- if $s = (R_0, \ldots, R_{m-1}, C)$ is state, then next($s$) is undefined if $C = n$ and otherwise is $s' = (R_0', \ldots, R_{m-1}', C')$ where
  - if $I_C = \text{INC}(i)$, then $C' = C + 1$ and $R_i' = R_i + 1$ and $R_k' = R_k$ for $k \neq i$;
  - if $I_C = \text{DEC}(i, j)$ and $R_i > 0$, then $C' = C + 1$ and $R_i' = R_i - 1$ and $R_k' = R_k$ for $k \neq i$;
  - if $I_C = \text{DEC}(i, j)$ and $R_i = 0$, then $C' = j$ and and $R_k' = R_k$ for all $k$.

We write $s \to s'$ if $s' = \text{next}(s)$.

An input for $M$ is a tuple $R = (R_0, \ldots, R_{m-1})$ of values for the registers. A run of $M$ on input $R$ is a sequence of states $(R_0, \ldots, R_{m-1}, 0) = s_0 \to s_1 \to \ldots$ such that either the sequence is infinite or the sequence is finite and the final state $s_f$ has no next state. The output of $M$ on input $R$ is the tuple $R'$ of register values in state $s_f$, or undefined if there is no final state.
2 Programming Register Machines

Writing programs for register machines is a very basic form of assembly language programming. As such, it is effectively impossible for most humans to write a non-trivial register machine program. We therefore introduce some extensions to the formalism to allow us to write programs in a slightly higher level language. These are explained quickly and informally on the slides. If you prefer formal definitions, they are here. Some of the choices I make here could be done differently, and/or are purely a matter of programming convention.

**Definition 3** An extended register machine program comprises:

- A set \( L \) of labels;
- a sequence \( I_0 \ldots I_{n-1} \) of extended instructions satisfying the constraint stated below;
- a labelling function \( \text{lab} : L \rightarrow \{0, \ldots, n\} \);
- a list \( D_0, \ldots, D_l \) of macro definitions

An extended instruction is

- \( \text{inc}(i) \) or \( \text{decjz}(i, j) \) where \( i \) is either an integer or a macro register parameter, and \( j \) is a non-negative integer, or a label, or a macro label parameter; or
- a macro invocation, which has the form \( \text{name}(i_1, \ldots, i_r, j_1, \ldots, j_s) \), where \( \text{name} \) is defined in the macro definition list, and if this instruction is part of a macro definition, then \( \text{name} \) is defined earlier in the list; \( i_1, \ldots, i_r \) and \( j_1, \ldots, j_s \) are as for \( i \) and \( j \) above, and \( r \) and \( s \) match the number of parameters in the definition.
- In either case, a macro parameter can only occur in the body of a macro definition, and must be one of the parameters of the definition.

A macro definition has the form \( \text{name}(\rho_1, \ldots, \rho_r, \lambda_1, \ldots, \lambda_s) = H_0 \ldots H_t \) where the \( \rho_k \) are the \( r \) register parameters, the \( \lambda_k \) are the \( s \) label parameters, and the \( H_k \) are extended instructions forming the body of the definition.

To make things easier to read when we write programs, we may adopt some common programming language conventions: instead of writing out the labelling function explicitly, we write a labelled instruction ‘label: \( I_k \)’ (in the main program only) to show that \( \text{lab}(\text{label}) = k \).

We shall also adopt some programming conventions. The definition above allows for registers with negative indexes (not with negative values!) as well as non-negative. Our convention is that negative-indexed registers are used only within macro bodies, and that furthermore every macro in a definition list must use different negative-indexed registers from any other macro. For example, the first macro might use \( R_{-1} \) and \( R_{-2} \) internally, the second macro might use none, the third \( R_{-3} \) and so on. Finally, we require that every macro be programmed to leave all its special registers at value zero when it exits (as defined below). Alternatively, we could say that every macro must start by clearing its special registers, which would in fact be easier and safer.

Additional considerations arise if one uses labels inside macro bodies (which is not covered by the above definitions). As there is no notion of scoped variables in our formalism, every use of a macro needs to use distinct labels in its body, or alternatively some form of scoping needs to be introduced. There are a number of ways of dealing with this – all are tedious, and this is not a course on language semantics and implementation. So we leave this as an exercise. Feel free to ask on Piazza if you want any further info or hints.

An extended register machine program is converted into an actual register machine by
replacing each macro invocation by the macro body, substituting the parameters appropriately. Because this process increases the number of instructions, we have to update all the branch instructions and label targets as well. The following formalizes this procedure.

**Definition 4** Given an extended register machine program

\((L, \text{lab}, D_0, \ldots, D_l, P = I_0 \ldots I_{n-1})\)

as above, its *assembled* register machine \(M'\) is defined as follows.

The registers of \(M'\) are all the registers mentioned in instructions of \(M\): either in the main program, or in macro bodies.

During the procedure we will maintain and change the labelling function \(\text{lab}\), a current offset \(o\), a relocation map \(\text{rel} : \{0, \ldots, n\} \to \mathbb{N}\), the extended program \(P\), and the generated program \(P'\). The offset \(o\) is initially zero, rel is initially the identity, and \(P'\) is initially empty.

While \(P\) is non-empty do:

- Let \(J_0 \ldots J_{p-1}\) and \(K_0 \ldots K_{p'-1}\) be the current contents of \(P\) and \(P'\) respectively.

According to the type of \(J_0\) do:

- If \(J_0\) is \(\text{inc}(i)\) let \(K_{p'} = J_0\), or if \(J_0\) is \(\text{decjz}(i, j)\) then let \(K_{p'} = \text{decjz}(i, j')\), where
  - if \(j\) is an integer, then \(j' = \text{rel}(j)\), or
  - if \(j\) is a label, then \(j' = \text{rel}(\text{lab}(j))\)
  add 1 to \(o\), and set \(P = J_1 \ldots J_{p-1}\) and \(P' = K_0 \ldots K_{p'}\).
- If \(J_0\) is \(\text{name}(i_1, \ldots, i_r, j_1, \ldots, j_s)\), and \(\text{name}\) has the definition \(H_0 \ldots H_t\), then
  - Let \(H'_k\) be the result of replacing each \(\rho_x\) by \(i_x\) and each \(\lambda_x\) by \(j_x\) in \(H_k\), and then adding \(o\) to every integer branch address or label parameter in the result.
  - set \(P = H'_0 \ldots H'_t J_1 \ldots J_{p-1}\);
  - update the relocation function as follows: if \(\text{rel}(x) = y\) and \(y > o\), then set \(\text{rel}(x)\) to be \(y + t - 1\).

It is an exercise for the reader to determine that this formal definition does what is intended – or alternatively, to find the bugs in it, of which there are almost certainly some. Note also that the resulting RM still has negative-indexed registers: it is a very simple exercise (do it) to convert a machine with negative-indexed registers to a standard RM as defined. Note also that I do **not** expect you to do or know any of this formal programming as an outcome of the course – the informal manipulation on the slides will be fine.