Introduction to Mathematical Logic

Martin Otto
Stephane Le Roux
Winter 2012/13

Solution Hints for Exercises No.9

Exercise 1 R-programs over alphabet $\{0,1\}$. For easier readability we use a GOTO construct, which is easiliy implemented by means of an "empty" branching condition of the form
IF $R_1 = \square$ THEN ℓ ELSE ℓ OR ℓ.

(a) $P_1 : R_2 := R_1^{-1}$ using registers R_1, R_2, R_3. The first part of this program moves the content of R_1 in reverse to both R_2 and R_3; the second part then moves the content of R_3 in reverse back to R_1.

0	IF $R_1 = \square$ THEN 9 ELSE 1 OR 5
1	$R_1 := R_1 - 0$
2	$R_2 := R_2 + 0$
3	$R_3 := R_3 + 0$
4	GOTO 0
5	$R_1 := R_1 - 1$
6	$R_2 := R_2 + 1$
7	$R_3 := R_3 + 1$
8	GOTO 0
9	IF $R_3 = \square$ THEN 16 ELSE 10 OR 13
10	$R_3 := R_3 - 0$
11	$R_1 := R_1 + 0$
12	GOTO 9
13	$R_3 := R_3 - 1$
14	$R_1 := R_1 + 1$
15	GOTO 9
16	STOP

(b) $P_2 : R_2 = R_1?$ using registers R_1, R_2, R_3. The main loop (lines $0-9$) reduces the register contents of both R_1 and R_2 by one if the last letters agree; or exits to termination with empty R_3 via line 11 if they do not agree; or exists to termination with non-empty R_3 via line 10, if both have been reduced to $\square$.

0	IF $R_1 = \square$ THEN 1 ELSE 2 OR 6
1	IF $R_2 = \square$ THEN 10 ELSE 11 OR 11
2	$R_1 := R_1 - 0$
3	IF $R_2 = \square$ THEN 11 ELSE 4 OR 11
4	$R_2 := R_2 - 0$
5	GOTO 0
6	$R_1 := R_1 - 1$
7	IF $R_2 = \square$ THEN 11 ELSE 11 OR 8
8	$R_2 := R_2 - 1$
9	GOTO 0
10	$R_3 := R_3 + 1$
11	STOP

Remark: a nicer version of this would either clean up R_1 and R_2 or restore these inputs.

Exercise 2 The following program assembles the reverse of the desired output in register R_2, leaving R_1 empty. [It should be combined with a program as in Exercise 1 (a) to reverse copy the contents of R_2 back into R_1.]

$R_3 = \square$ or $R_3 = 1$ serves as a marker: $R_3 = 1$ until the first 0 from R_1 is processed.

0	$R_3 := R_3 + 1$	[initialize R_3]
1	IF $R_3 = \square$ THEN 2 ELSE 9 OR 9	[only $R_3 = \square$ or $R_3 = 1$ occurs]
2	IF $R_1 = \square$ THEN 17 ELSE 3 OR 6	[branching on R_1, while $R_3 = \square$]
3	$R_1 := R_1 - 0$	[if $R_1 = \ldots 0$, $R_3 = \square$]
4	$R_2 := R_2 + 0$	
5	GOTO 1	
6	$R_1 := R_1 - 1$	[if $R_1 = \ldots 1$, $R_3 = \square$]
7	$R_2 := R_2 + 1$	
8	GOTO 1	
9	IF $R_1 = \square$ THEN 11 ELSE 10 OR 14	[branching on R_1, while $R_3 = 1$]
10	$R_1 := R_1 - 0$	[if $R_1 = \ldots 0$, $R_3 = 1$]
11	$R_2 := R_2 + 1$	
12	$R_3 := R_3 - 1$	
13	GOTO 1	
14	$R_1 := R_1 - 1$	[if $R_1 = \ldots 1$, $R_3 = 1$]
15	$R_2 := R_2 + 0$	
16	GOTO 1	
17	STOP	

Exercise 3 (i) $\Rightarrow$ (ii). Assume W is decidable and infinite. The unique monotone function $f\colon \mathbb{N} \to W$ (mapping n to the $n+1$-st element of W w.r.t. $<_{\text{lex}}$) is surjective. It is also computable: to determine $f(n)$ one needs to run a decision procedure for W successively stepping through $w \in \mathbb{A}^*$ in lexicographic order, until one gets the $n+1$-st positive answer, and then output the corresponding w.

(ii) $\Rightarrow$ (i). W.l.o.g. consider an infinite W with f as given. The following algorithm then decides whether $w \in W$: compute successive values $f(n)$ for $n = 0, 1, 2, \ldots$ until either $f(n) = w$ (answer: $w \in W$) or $w <_{\text{lex}} f(n)$ (answer: $w \notin W$).

Exercise 4 (cf. Lemma 4.1.2) For (i) $\Rightarrow$ (ii) one just needs to modify the tail end of a program P that decides W so that instead of terminating with $R_1 = \square$ say, it diverges. For this it suffices to replace the last line "m STOP" by the lines

m	IF $R_1 = \square$ THEN m ELSE $m+1$ OR$\ldots$ OR $m+1$
$m+1$	STOP

The new programm P' terminates on input u iff P terminates with nonempty R_1 on input u, and it diverges if P terminates with empty R_1 on input u. So $W = \{u\colon u \xrightarrow{P'} \text{STOP}\}$ is r.e.
$\overline{W}$ is treated analogously.

For (ii) $\Rightarrow$ (i) we assume P_1 and P_0 are such that $W = \{w \in \mathbb{A}^* \colon w \xrightarrow{P_1} \mathsf{STOP}\}$ and $\overline{W} = \{w \in \mathbb{A}^* \colon w \xrightarrow{P_0} \mathsf{STOP}\}$. Note that on each input $w \in \mathbb{A}^*$, precisely one of P_1 or P_0 terminates. W.l.o.g. assume that P_0 and P_1 use disjoint sets of registers (also for their input), and that R_1 and R_2 is used by neither.

Now organise P such that an initial part (taking the first m lines) serves to copy the register content of its input register R_1 to the two input registers of P_1 and P_0.

For the rest of P we want it to simulate P_0 and P_1 in an interleaving fashion, executing appropriate lines of P_0 and of P_1 in alternating manner, until one of them stops. For this we use pieces of program (one or two lines of instructions each) labelled by pairs of line numbers with a marker on either the first or the second component: $(\underline{\ell_0}, \ell_1)$ and $(\ell_0, \underline{\ell_1})$.

If line ℓ of P_i is a register update command α, then let

$$(\underline{\ell}, \ell_1) : \begin{cases} \alpha \\ \mathsf{GOTO}\ (\ell+1, \underline{\ell_1}) \end{cases} \qquad\qquad (\ell_0, \underline{\ell}) : \begin{cases} \alpha \\ \mathsf{GOTO}\ (\underline{\ell_0}, \ell+1) \end{cases}$$

If line ℓ of P_i is a conditional branching command α, let $\alpha'(\ell_1)$ be the result of replacing each line address ℓ' in α by $(\ell', \underline{\ell_1})$; similarly let $\alpha''(\ell_0)$ the result of replacing each line address ℓ' in α by $(\underline{\ell_0}, \ell')$. Then put

$$(\underline{\ell}, \ell_1) : \quad \alpha'(\ell_1) \qquad\qquad (\ell_0, \underline{\ell}) : \quad \alpha''(\ell_0)$$

If line ℓ of P_i is the STOP line, put

$$(\underline{\ell}, \ell_1) : \begin{cases} R_2 := R_2 + a \\ \mathsf{GOTO}\ L \end{cases} \qquad\qquad (\ell_0, \underline{\ell}) : \quad \mathsf{GOTO}\ L$$

Finally make L the new STOP line of P.

With a suitable renumbering of the new program lines and transcription of the GOTO command (see above), P is turned into a proper R-program with input register R_1 and output register R_2 for deciding W: P terminates on all inputs u, and terminates with empty output register R_2 iff $u \notin W$ (viz., if P_1 terminates on u).

Exercise 5 (a) There is no general implication since the languages $\emptyset$ and $\mathbb{A}^*$ are both decidable.

(b) For example, assume that both L_1 and L_2 are enumerable by programs P_1 and P_2, respectively. Then $L_1 \cup L_2$ is also enumerable. Indeed take the input and feed it to both programs running in parallel. As soon as one of the program stops (thus acknowledging membership of the input), stop the other program. For $L_1 \cap L_2$ we would have to wait that both programs stop.

(c) First solution: there are only countably many enumerable languages (since programs are countably many) but there are uncountably many languages L such that $L_1 \subseteq L \subseteq L_2$. Second solution, more constructive: In order to build such an undecidable language, consider a numbering of all the words in $L_2 \backslash L_1$ and a numbering of all the register programs, *e.g.*, over the binary alphabet. Define L as the union of L_1 plus the words in $L_2 \backslash L_1$ whose numbers correspond to a program that halts. The halting problem is reducible to membership in L, which is then undecidable.

Exercise 6 Following the hint, we want to describe the operation of P on register content $\overline{u} = (u_1, u_2, \ldots, u_k)$ in terms of $\overline{u}' = (u_1 \# u_2 \cdots \# u_k, \Box)$. For the initial configuration on input $u \in \mathbb{A}^*$, we transform the initial register content $(u, \Box, \ldots, \Box)$ into $\overline{u}'_0 = (u\#\# \cdots \#, \Box)$ by a string of $k-1$ lines with the command $R_1 := R_1 + \#$.

A command $R_i := R_i \pm a$ in P now has to be simulated by the following subroutine: reverse-copy the top of the content of R_1 down through the first $(k-i)$ occurrences of $\#$ from the top (do nothing if $i = k$) so that the register content of P's register R_i is at the top of register R_1. Then operate as required on the remaining register content in R_1 ($R_i := R_i \pm a$ becomes $R_1 := R_1 \pm a$). Finally reverse-copy the content of R_2 back onto R_1.

An $\mathsf{IF}\ R_i = \Box\ \mathsf{THEN}\ n_1\ \mathsf{ELSE}\ \ldots\ \mathsf{OR}\ n_\ell$ similarly needs to be prefixed by a reverse-copy operation that brings the register content of P's register R_i to the top of register R_1. The actual branching is then performed on $\mathsf{IF}\ R_1 = \Box\ \mathsf{THEN}\ n'_1\ \mathsf{ELSE}\ \ldots\ \mathsf{OR}\ n'_\ell\ \mathsf{OR}\ n'_1$ (note that the ELSE-selection is one longer than before, with a last OR for the case that the register contents ends in $\#$, which means that P's R_i was empty. Each one of the target lines $n'_1, \ldots, n'_\ell$ in P needs to be prefixed by a reverse-copy procedure that restores the upper layers of R_1.

NB: unlike the reverse-copy operation of Exercise 1 (a), we here do not want to keep the source register unchanged, but really remove letter after letter in the source register as we append it to the target register. This is easily done without any auxiliary registers.