Programming Languages

Qualifying Exam, January 1995

Read the whole exam before beginning to answer the questions.

This test revolves around Turing Machines (TMs) —we use TMs as a vehicle to explore issues of signatures, type, functional definitions, rewrite semantics, evaluation semantics, transition semantics, partial correctness, and compilation. The test is graded on a total of 100 points.

Use the following terminology and notation.

A Turing machine (TM) over a fixed, finite *tape alphabet* Σ (that contains the blank character) is a triple consisting of

1. S, a finite set of states ,
2. fs, the set of final states ($fs \subseteq S$)
3. A transition function δ from a state and a symbol to

   (1) a move in $Dir = \{-1, 0, +1\}$,
   (2) a new symbol (in Σ) to be printed, and
   (3) the next state.

A *tape* over Σ is a two-way infinite sequence $\sigma = \ldots \sigma_{-2}, \sigma_{-1}, \sigma_0, \sigma_1, \sigma_2, \ldots$ of elements in Σ, of which only a finite number are nonblank. Note that we use two-way tapes. The notation $(\sigma; i : a)$ denotes a tape that is a copy of σ except that at position i its value is a.

An *instantaneous description* id over Σ consists of a state, a tape, and an integer showing the position of the read/write head. Use the notation id_1, id_2, id_3 to refer to the three components of id. Let ID denote the set of instantaneous descriptions.

A *move* of a TM is a mapping of an id to the next id, the latter being obtained by advancing the head, then writing a new symbol, and finally changing the state according to δ. Give the name $\hat{\delta}$ to this function $\hat{\delta} : ID \rightarrow ID$ on instantaneous descriptions.

Question 1 (15 points). Consider a fixed alphabet Σ. Let *DFinite* denote the type of *discrete finite* sets, i.e. the finite sets for which equality is decidable.

(a) Write a signature (as a dependent product type or an ML signature) for the type of TMs, e.g.

$$TM = \_\_ \times \_\_ \times \_\_$$

(b) Define the type of legal instantaneous descriptions. In doing so, use $TAPE$ for the type of all tapes over Σ. Note that a tape is an infinite entity, so that it might seem that $TAPE$ cannot really be defined in a language like ML. However, a tape can be described by a finite entity since all but a finite number of symbols on it are nonblank. Describe a data structure for implementing tapes, thus showing that $TAPE$ can be defined in ML.

(c) Define function $\hat{\delta} : ID \to ID$ of a TM (s, fs, δ), using ML or any other functional notation of your choosing. You may assume (throughout) that subscripting is already defined.

(d) Define a generating function *Compute* that accepts an initial *id* and produces a *computation*, i.e. a stream of *id*s in the type

$$Comp = 1 + ID \times Comp \qquad \text{e.g. } \nu(C \,.\, 1 + ID \times C)$$

Here, unit type 1 is used to signal a halting computation. Use a functional notation of your choice. If you use CS611 stream functions, recall that the stream $\langle d, d+1, d+2, \ldots \rangle$ is generated by $\nu\_ind(d;\ z, f.\langle z, f(z+1)\rangle)$.

Question 2 (10 points). Transition function $\hat{\delta}$ (see question 1(c)) defines a rewrite semantics (reduction semantics, according to Gunter) for TMs, where the steps are $id \to id'$ iff $\hat{\delta}(id) = id'$. We say that the $\hat{\delta}$ rewrite semantics for *id* yields id' in the case that there exists instantaneous descriptions $id0, \ldots, idn$ such that $idn_1 \in fs$, $idj_1 \notin fs$ for j in $0..n-1$, and

$$id = id0 \to id1 \to \ldots \to idn = id'$$

(a) Define inductively a relation $id \downarrow id'$ that gives an evaluation semantics for TMs, that is a TM computing from *id* halts in id'.

(b) Sketch a proof that if the $\hat{\delta}$ rewrite semantics for *id* yields id', then your evaluation semantics concludes $id \downarrow id'$. Be clear about any inductions you use.

(c) In CS611, we gave reduction semantics $\to$ and evaluation semantics $\downarrow$ for the untyped lambda calculus and proved a result analagous to part 3(b). The proof of 3(b) is much easier, though. Explain why it is much easier.

Question 3 (25 points). An *alternating TM* is a 3-tuple

$$\begin{aligned} M = (&S, && \text{(state space)} \\ &fs = \{s_{\text{yes}}, s_{\text{no}}\} && \text{(set of final states)} \\ &\delta : S \times \Sigma \to Dir \times \Sigma \times B_S \\ ) & \end{aligned}$$

Here, final state s_{yes} gives the answer *YES* and final state s_{no} gives the answer *NO*. Transition function δ maps a state and a symbol into a move, an output symbol, and boolean expression in B_S. B_S is the set of boolean expressions that contain only elements in S as variables and binary operations $\wedge$ (and) and $\vee$ (or). For example, one such expression is $(s_{\text{yes}} \vee s_0) \wedge s_1$.

An instantaneous description $id = (b, \sigma, i)$ is as in the preamble to the test, except that the first component is a boolean expression in B_S instead of S.

The evaluation semantics of *id*, given in Fig. 1, indicates the result $V \in \{YES, NO\}$ of starting machine M in instantaneous description *id*. Here is an explanation of rule (3). Instantaneous

Table 1: EVALUATION SEMANTICS FOR ALTERNATING TM

(1) $(s_{\text{yes}}, \sigma, i) \downarrow YES$

(2) $(s_{\text{no}}, \sigma, i) \downarrow NO$

(3) For $s \in S - \{s_{\text{yes}}, s_{\text{no}}\}$,

$$\frac{(b,\ \ (\sigma; i+d : a),\ \ i+d) \downarrow V}{(s, \sigma, i) \downarrow V} \qquad \text{where } \delta(s, \sigma_i) = (d, a, b)$$

(4) $$\frac{(b_1, \sigma, i) \downarrow V}{(b_1 \vee b_2, \sigma, i) \downarrow V} \qquad \frac{(b_2, \sigma, i) \downarrow V}{(b_1 \vee b_2, \sigma, i) \downarrow V}$$

(5) $$\frac{(b_1, \sigma, i) \downarrow V,\ \ (b_2, \sigma, i) \downarrow V}{(b_1 \wedge b_2, \sigma, i) \downarrow V}$$

description (s, σ, i) computes to V (this is what the conclusion states) if id' computes to V (this is what the premise states), where id' is the result of making the step given by transition function $\delta(s, \sigma_i)$.

Instantaneous description $(s_1 \vee s_2, \sigma, i)$ computes to V if either (s_1, σ, i) or (s_2, σ, i) computes to V —so this is nondeterminisitic choice. Instantaneous description $(s_1 \wedge s_2, \sigma, i)$ computes to V if both (s_1, σ, i) and (s_2, σ, i) compute to V.

This semantics is naturally extended to arbitrary boolean expressions (rules (4) and (5)).

Your job is to implement M, by giving a transition semantics $\rightarrow$ such that the following holds. (The set of configurations may be anything you wish, for example (and this is only an example), you may wish to use a binary tree of instantaneous descriptions as a state.)

(a) $(s, \sigma, i) \rightarrow^+ V$ iff $(s, \sigma, i) \downarrow V$.
(b) Every transition step processes at most one tape symbol.

Question 4 (20 points). A TM (see the definition before Question 1) can be viewed as an imperative program that manipulates three variables that constitute an instantaneous description: s (the current state), σ (the current tape), and i (the current position of the head). Viewed in this way, we can imagine defining the partial correctness of a TM. Let Q and R be predicates over variables s, σ, i. Let the notation

$$\{Q\}\ TM\ \{R\}$$

mean "if execution of TM begun with Q true terminates, then it terminates with R true.". Note that we are using the notation $\{Q\}\ TM\ \{R\}$ to denote partial correctness, not total correctness.

Your task is to develop a set of conditions, written in predicate-calculus notation, that together imply $\{Q\}\ TM\ \{R\}$, for general predicates Q and R. These requirements should be in terms of Q, R, δ, fs, s, σ, and i (and any other predicates that you need).

We suggest that you do this by first writing the TM as an imperative program —but the final answer should not be in terms of program statements.

Question 5 (30 points). Let PL/TM be a "programming language" for defining and executing deterministic Turing machines with 2-way infinite tapes. In PL/TM,

- Alphabet Σ consists of an initial sequence of lower-case letters $\mathtt{a}, \ldots, \ell$ and _ (denoting the blank tape symbol).
- States S are represented by natural numbers $0, \ldots, n$, where the initial state is 0 and the final state is n.
- Transition function δ is defined by a sequence of zero or more 5-tuples ⟨ *state, letter-or-underscore, state, move, letter*⟩. Note that _ cannot be written on the tape.
- Moves are represented by L (left), . (no move), and R (right).

A "program" in PL/TM consists of a letter ℓ (signifying that $\Sigma = \{a, \ldots, \ell\}$), a natural number n (signifying that $S = \{0, \ldots, n\}$), and zero or more 5-tuples (defining δ). Whitespace characters newline and space can appear anywhere in the program except within a number. For example, the following is a PL/TM program:

$$b\ \ 2\ \ \langle 0, a, 0, R, a\rangle\ \ \langle 0, b, 1, R, b\rangle\ \ \langle 1, \_, 2, ., b\rangle$$

If the initial input tape (provided by the user at the beginning of execution) is accepted, the program prints `Yes`, otherwise it either prints `No` or runs forever.

Choose a target object language OL from among { C, Pascal, some assembler } and write a compiler that translates PL/TM to OL. Use any implementation language and compiler development tools you wish. For example, you could generate C using C, *lex*, and *yacc*. Use pseudo code or prose where you feel it is appropriate to omit obvious details, but be precise and complete. Our solution is about 2 typeset pages.

(a) Define the lexical structure of PL/TM with regular expressions and define its context-free syntax with an unambiguous LL(1) or LR(1) grammar.

(b) Write a scanner/parser for PL/TM modeled on your part (a) answer. If you use scan/parser tools, just argue that they apply to your part (a) answer.

(c) Augment your scanner/parser to do appropriate "static semantic analysis" for PL/TM.

(d) Describe your target "run-time environment" and "storage management scheme".

(e) Augment your scanner/parser/analyzer to generate the OL code.