Advanced Computer Science 2

Final exam, May 25, 2004

Note: a maximum of 100 points is accredited for this exam.

Groups A & B with solutions

Problem 1 (15 or 30 points). The single-tape TMs that we considered could decide languages, that is sets of finite strings, that is, sets of finite one-dimensional patterns. One can also consider 2-D languages that consist of sets of finite two-dimensional patterns. Such a language might for instance look like

$$L_{2D} = \left\{ \begin{array}{|c|c|c|}\hline 0 & 1 & 1 \\ \hline 1 & 0 & 1 \\ \hline \end{array} \;,\quad \begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline \end{array} \;,\quad \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 1 & 0 \\ \hline 1 & 1 \\ \hline \end{array} \right\}$$

Give a *formal* definition of 2-D languages and 2-D TMs that can operate on 2-D input patterns. Note: there are many ways of defining these items (just as there are many ways to define standard TMs). Think out one alternative – you are quite free here – but make sure that your definitions are consistent and complete. Additional note: there are two basic approaches: (i) first define 2-D languages, then define how their "words" can be converted to standard 1D words, then just use standard TMs; (ii) define 2-D languages and devise a truly 2D-version of TMs that uses a 2D-"checkerboard" data carrier instead of a 1D tape. Approach (i) brings 15 points, approach (ii) 30 points. Choose only one of the approaches.

Solution: (option ii) *Definition (2D-languages):* Let Σ be an alphabet. A *rectangle* is a pair $R = (n,m)$, where n, m are integers ≥ 1. A *2D-word* over Σ is a pair (R, f), where $R = (n,m)$ is a rectangle and f: $\{1,...,n\} \times \{1,...,m\} \to \Sigma$. A *2D-language over* Σ is a set of 2D-words over Σ.

Note: something like this suffices – it is consistent and formal. One might want to embellish this definition in various ways (for instance, define empty word or concatenation of words), but for the purpose of the exam problem this would suffice. Such add-ons would give extra points.

Definition (2D-TMs): A 2D-Turing machine is a structure $M = (K, \Sigma, \delta, s)$, where K is a finite set of *states*, $s \in K$ is the *initial state*, the *alphabet* Σ is a set of (tape) *symbols*, and where K and Σ are disjoint. We assume that Σ always contains the special symbols $\sqcup$ and $\rhd$ and $\triangledown$ and $\bigstar$, the *blank* and the *left margin* and the *upper margin* and the *corner* symbol. Finally, δ is a *transition function*, where

$$\delta: K \times \Sigma \to (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, \uparrow, \downarrow, -\}.$$

We assume that h (the *halting state*), "yes" (the *accepting state*), "no" (the *rejecting state*), and the *cursor directions* ←, →, ↑, ↓, –, are extra symbols not in $K \cup \Sigma$.

Note: this is the bare definition of (one version of) a 2D-TM. Something like this would be enough for the exam problem. In further definitions one would define configurations, transitions, input presentation, computation results, etc. Such add-ons give extra points in the exam (if they are correctly formulated).

Note: there are many ways to define such languages and their TMs. The solution shown here should just indicate the degree of formal rigour expected.

Problem 2 (20 points). Give an example of a language that cannot be accepted by a TM, and state why it can't.

Solution: There are uncountably many languages that cannot be accepted by a TM, so there are very many solutions... For instance, H^c cannot be accepted. Proof by contradiction. Assume H^c is acceptable. We know H is acceptable. If a langage and its complement are acceptable, both are decidable. But H is undecidable. Therefore H^c is not acceptable.

Problem 3 (20 points). How many languages over {0,1}* are undecidable? finitely many, countably many, or uncountably many? Why?

Solution: There are uncountably many languages over {0,1}* but only countably many TMs (modolo computational equivalence) exist. Therefore at only countably many languages can be decided. Because "uncountable minus countable is uncountable", there exist uncountably many undecidable languages.

Problem 3 (20 points). How many functions f: $\mathbb{N} \to \{0,1\}$ are not recursive? finitely many, countably many, or uncountably many? Why?

Solution: There are uncountably many functions from $\mathbb{N}$ to {0,1} [regardless of whether you consider totally or partially defined functions] but only countably many TMs (modolo computational equivalence) exist. Therefore at only countably many (partial or total) functions are recursive. Because "uncountable minus countable is uncountable", there exist uncountably many non-recursive languages.

Problem 4 (20 points). If $[x_1, ..., x_k] = (x_1, (x_2, (.... (x_{k-1}, (x_k, \mathbf{nil})) ...)))$ and $[y_1, ..., y_l] = (y_1, (y_2, (.... (y_{l-1}, (y_l, \mathbf{nil})) ...)))$ are two lists, the *join* of them is the list $[x_1, ..., x_k, y_1, ..., y_l] = (x_1, (x_2, (.... (x_{k-1}, (x_k, (y_1, (y_2, (.... (y_{l-1}, (y_l, \mathbf{nil})) ...))))) ...)))$. Define a λ-expression **join** that if called with two lists returns the join of them. You may use the test **nul**, that is, a λ-expression that returns **true** if its argument is the empty list and **false** if it is a non-empty list, and you may use **if** and **cons**, and you may use **last**, which returns the last non-nil element of a list, and you may use **beginning**, which returns a list without its last non-nil element.

Solution: Intuitively, one would want to define **join** by the recursion

$$\mathbf{join}\ L_1\ L_2\ =\ \mathbf{if}\ (\mathbf{nul}\ L_1)\ L_2\ (\mathbf{join}\ (\mathbf{beginning}\ L_1)\ (\mathbf{cons}\ (\mathbf{last}\ L_1)\ L_2))$$

This is of course not an admissible definition, because the λ-expression **join** that we want to be defined occurs on the right hand side. Using Y, we obtain the correct definition

$$\mathbf{join}\ \equiv\ Y(\lambda\ g\ L_1\ L_2\ \mathbf{if}\ (\mathbf{nul}\ L_1)\ L_2\ (g\ (\mathbf{beginning}\ L_1)\ (\mathbf{cons}\ (\mathbf{last}\ L_1)\ L_2)))$$

Problem 4 (20 points). If $[x_1, ..., x_k] = (x_1, (x_2, (.... (x_{k-1}, (x_k, \textbf{nil})) ...)))$ and $[y_1, ..., y_l] = (y_1, (y_2, (.... (y_{l-1}, (y_l, \textbf{nil})) ...)))$ are two lists, the *join* of them is the list $[x_1, ..., x_k, y_1, ..., y_l] = (x_1, (x_2, (.... (x_{k-1}, (x_k, (y_1, (y_2, (.... (y_{l-1}, (y_l, \textbf{nil})) ...))))) ...)))$. Define a λ-expression **join** that if called with two lists returns the join of them. You may use the test **nul**, that is, a λ-expression that returns **true** if its argument is the empty list and **false** if it is a non-empty list, and you may use **if** and **cons** and **head** and **tail**, and you may use **reverse**, which reverses lists.

Solution: First, define **last**, which returns the last non-nil element of a list, and **beginning**, which returns a list without its last non-nil element, through **reverse**:

beginning $\equiv$ $\lambda\, L$ (**reverse** (**tail** (**reverse** L)))
last $\equiv$ $\lambda\, L$ (**head** (**reverse** L)).

Next, intuitively, you would want to define **join** by the recursion

join L_1 L_2 = **if** (**nul** L_1) L_2 (**join** (**beginning** L_1) (**cons** (**last** L_1) L_2))

This is of course not an admissible definition, because the λ-expression **join** that we want to be defined occurs on the right hand side. Using Y, we obtain the correct definition

join $\equiv$ $Y(\lambda\, g\, L_1\, L_2$ **if** (**nul** L_1) L_2 (g (**beginning** L_1) (**cons** (**last** L_1) L_2)))

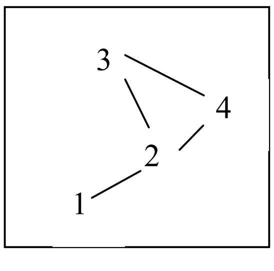
NOTE: after writing the model solutions as above (and putting them online), I learnt from an exam solution a simpler solution that makes do with just **head** and **tail:**

join $\equiv$ $Y(\lambda\, g\, L_1\, L_2$ **if** (**nul** L_1) L_2 (**cons** (**head** L_1) (g (**tail** L_1) L_2)))

Problem 5 (20 points). Relate the two statements "computers can't think" and "**P** ≠ **NP**" to each other. Format of solution: what I would like to see here is a short (half page) essay that can be witty and need not be formal.

Solution: Comment: What I thought of when I formulated this problem was that **P** ≠ **NP** is related to human, insightful reasoning through the "guessing" part in nondeterministic TMs. If **P** = **NP**, then the "guessing of a correct solution" can be carried out mechanically by a computer in reasonable time – that is, the "insightful", "creative" aspect of human reasoning turns out to be something "mechanical". This is a good core idea for a little essay. I am curious to see what *you* wrote about.

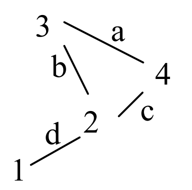
Problem 6 (40 points). Consider the two problems VERTEX COVER and FEEDBACK ARC SET (FAS) [for problem specification see below]. VERTEX COVER is **NP**-complete. Show that FAS is also **NP**-complete. Hint: turn edges into cycles! Components of solution: (a, 10 points) Describe in one sentence how one typically goes about to show **NP-**completeness, and in another sentence why this works. (b, 20 points) Describe informally a method to reduce VERTEX COVER to FEEDBACK ARC SET. (c, 10 points) Show how the following instance of VERTEX COVER looks after the transformation:



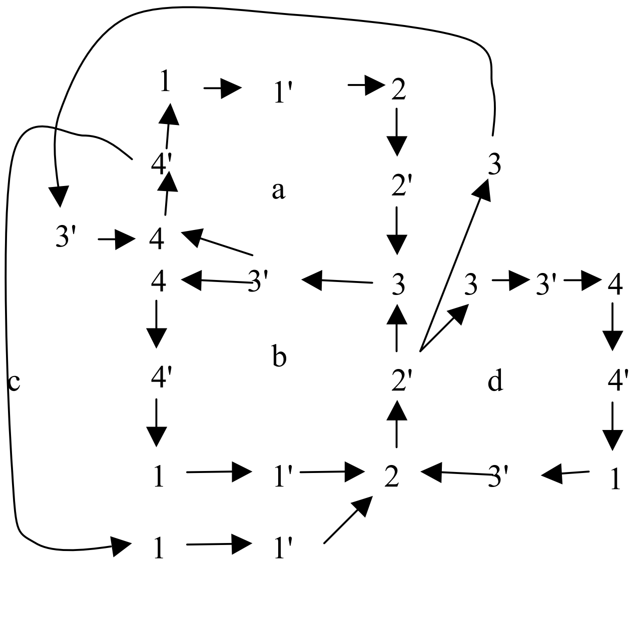
Specification of the two problems: An instance of the problem FEEDBACK ARC SET (FAS) is a directed graph $G = (V, A)$ [where V is the set of nodes and A the set of directed edges], plus a constant K. The question: is there a subset $A' \subseteq A$ [called a feedback arc set] where $|A'| \leq K$, such that A' contains at least one edge from every directed cycle in G? (a directed cycle is a subgraph $v_1 \to v_2 \to ... \to v_n \to v_1$, where $n \geq 1$ and all v_i are pairwise different). —— An instance of VERTEX COVER is an undirected graph $H = (W, B)$ plus a constant L. The question: is there a subset $W' \subseteq W$, where $|W'| \leq L$, such that for every edge $\{u,v\} \in B$, at least one of u and v belongs to W'?

Solution (more formal than required in exam): We have to transform instances $x = (W, B)$, L of VERTEX COVER into instances $R(x) = (V, A)$, K of FAS, such that this transformation takes at most polynomial time and that the FAS answer on the instance $R(x)$ is identical to the VERTEX COVER answer to x. One way of doing so (in fact, the only way I could think of) is to create (V, A) in three steps. Let $W = \{w_1, w_2, ... w_N\}$, and enumerate the edges in B by $e_1, ..., e_M$. Step 1: create a preliminary node set V_0 by introducing, for every edge $e_i \in B$, $2N$ nodes $v_{i1}, v'_{i1}, v_{i2}, v'_{i2}, ... v_{iN}, v_{iN}$. Step 2: Establish on V_0 a preliminary edge set A_0, by connecting all the groups $v_{i1}, v'_{i1}, v_{i2}, v'_{i2}, ... v_{iN}, v_{iN}$ into cycles, that is, add directed connections $v_{i1} \to v'_{i1} \to v_{i2} \to ... \to v_{iN} \to v'_{iN} \to v_{i1}$ to A_0. Step 3: for each node $w_j \in W$, consider the n nodes $w_{j'} \in W$ that are connected to w_j. Let the connections between w_j and the $w_{j'}$ be $e_{a_1}, ..., e_{a_n}$. Take the n cycles $v_{a_1 1} \to v'_{a_1 1} \to ... \to v'_{a_1 N} \to v_{a_1 1}, ..., v_{a_n 1} \to v'_{a_n 1} \to ... \to v'_{a_n N} \to v_{a_n 1}$ and identify across them the links $v_{a_1 j} \to v'_{a_1 j}, ..., v_{a_n j} \to v'_{a_n j}$, by merging $v_{a_1 j}$ with $v_{a_2 j}$ with $v_{a_3 j}$ ... and $v'_{a_1 j}$ with $v'_{a_2 j}$ with $v'_{a_3 j}$ etc. After doing this for all $w_j \in W$, the resulting merged node set and graph is (V, A). Finally, put $K = L$. It is clear from this construction that if in the context of VERTEX COVER, a node w_j has connections $e_{a_1}, ..., e_{a_n}$, then (only) the merged arc $v_{a_1 j} \to v'_{a_1 j} = ... = v_{a_n j} \to v'_{a_n j}$ in (V, A) is common to the (merged) cycles $v_{a_1 1} \to v'_{a_1 1} \to ... \to v'_{a_1 N} \to v_{a_1 1}, ..., v_{a_n 1} \to v'_{a_n 1} \to ... \to v'_{a_n N} \to v_{a_n 1}$, and furthermore, the only arcs that are common to several of these cycles correspond to nodes w_j. Therefore, a vertex cover of size L in (W, B) corresponds to a feedback arc set in (V, A). The transformation at most twice-squares the number of nodes and all steps in the construction can clearly be done in polynomial time per constructed node.

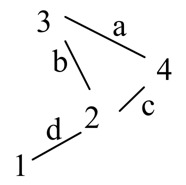
Graphical example:



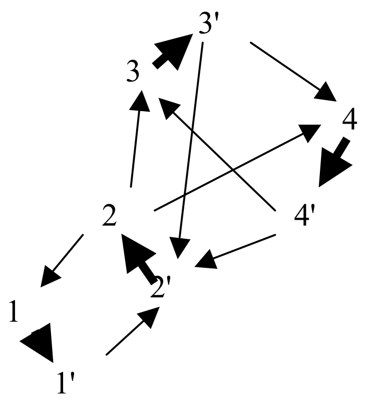
becomes



Second (much more elegant) solution, found by R. Rathnam in the exam!! Ravi's solution is wonderful and self-explaining from the example:



becomes



Computability and Complexity, Fall 2006: Final Exam – Solutions

Notes: A maximum of 100 points is accredited for this exam (sum of points from all problems is 115) . Points reflect difficulty of the problem (as I see it). All problems can be solved using no more than 10 lines of (printed) text each. Best wishes!

Problem 1 (25 points). Show that the language

L = {<M>;<N>;w | <M> is the coding of a deterministic TM M with input alphabet {0,1}; <N> is the coding of a deterministic TM N with input alphabet {0,1}; $w \in \{0,1\}^*$; $M(w)$ = "yes" and $N(w)$ = "yes"}

is undecidable, by reduction to the halting problem. (Assume that some coding convention is fixed).

Solution. Assume L is decidable. Let N_0 be a TM which accepts every input with "yes" (such an N_0 is easy to construct). Then we can reduce L_1 = {<M>;w | <M> is the coding of a deterministic TM M with input alphabet {0,1}; $w \in \{0,1\}^*$; $M(w)$ = "yes"} to L by <M>;w $\in L_1$ iff <M>;<N_0>;$w \in L$. Furthermore, we can reduce the halting language H = { <K>;x | $K(x) \neq \nearrow$} to L_1 by renaming the "yes", "no" and "halt" states of K into states "yes1", "no1", "halt1" and adding new rules ("yes1", #) → ("yes", #, –), ("no1", #) → ("yes", #, –),("halt1", #) → ("yes", #, –), obtaining K'; we then have <K>;x $\in H$ iff <K'>;x $\in L_1$. Concatenation of the two reductions yields <K>;x $\in H$ iff < K'>;<N_0>;$w \in L$, that is, we could decide H, contradiction.

Problem 2 (15 points). Show that the factorial function defined by $fac(0) = 1$, $fac(n+1) = (n+1) fac(n)$ is primitive recursive, by constructing it from the axioms and rules in Definition 5.1 in the lecture notes. You may assume that the function $prod(n, m) = nm$ is p.r.

Solution. One way to solve this task is to first derive that a function *prodplus*$(n, m) = (n + 1)$ m is p.r. Clearly *prodplus*(n, m) = *prod*$(\sigma(n), m)$. Using rule 4, this is verified to be p.r. by setting $f: \mathbb{N}^2 \to \mathbb{N}$, $f(m, n)$ = *prod*(m, n) and $g_1: \mathbb{N}^2 \to \mathbb{N}$, $g_1(m, n) = \sigma(p^2{}_1(m, n)) = \sigma(n)$ [needs itself a justification via rule 4, if one were painstakingly precise], $g_2(m, n) = p^2{}_2(m, n) = n$; then *prod*$(\sigma(n), m) = f(g_1(m, n), g_2(m, n))$.

Then essentially we have to apply the primitive recursion rule 5. For f and g in this rule, use f: $\mathbb{N}^0 \to \mathbb{N}$, $f() = 1$, which is p.r. by $f = \sigma\mathbf{0}$ (needs axioms 1, 2, 4). Furthermore, for g: $\mathbb{N}^2 \to \mathbb{N}$ use *prodplus*. Then rule 5 says that there exists a p.r. function h: $\mathbb{N} \to \mathbb{N}$, satisfying

(i) $h(0) = f() = 1$ and (ii) $h(n+1) = g(n, h(n)) = (n+1)\, h(n)$,

which is just the requirement for *fac*.

Problem 3 (30 points). Define a λ-expression, which when called with a list x_1 ::.... :: x_n :: **nil**, returns the 1-cycled list x_2 ::.... :: x_n :: x_1 :: **nil**.

Solution. One solution that I found late at night (your solutions will be more elegant I predict) is to exploit the join operator, a λ-expression which, when called with two lists $a = x_1$ ::.... :: x_n :: **nil**, $b = y_1$ ::.... :: y_m :: **nil**, returns the joined list x_1 ::.... :: x_n :: y_1 ::.... :: y_m :: **nil**. One way to define such a join operator turns the following naive recursion

```
join l m =  if (null l)
                m
                (if (null m)
                     l
                     (if (null (second l))
                          (cons (head l) m)
                          (cons (head l) (join (tail l) m))))
```

into this λ–expression, using the Y combinator:

```
Y(λ g l m. if (null l)
                m
                (if (null m)
                     l
                     (if (null (second l))
                          (cons (head l) m)
                          (cons (head l) (g (tail l) m))))
```

Using shorthand **join** for this λ–expression, we can obtain a λ–expression for the cycle-operation by

λ *l*. **join** (**tail** *l*) (**cons** (**head** *l*) **nil**)

Problem 4 (15 points). The instances of the problem PARTITION are finite sets $A = \{a_1, ..., a_n\}$, where for each $a \in A$ there a positive integer $w(a)$ is specified (the *weight* of a). The question is whether there exists a subset $A' \subseteq A$ such that $\sum_{a \in A'} w(a) = \sum_{a \in A-A'} w(a)$. Furthermore, for any instance $I = (A, w)$ define the *size* of the instance to $s(I) = |A| * \max\{\log(w(a)) \mid a \in A\}$. Specify a method to code instances of this problem into 0-1-strings. Verify that your coding method is linear in instance size, that is, $|\text{code}(I)| = O(s(I))$.

Solution. There are so many ways of doing this that a model solution makes little sense. One aspect of this problem that makes it a little tricky is the requirement that the coding results should be bitstrings, which is not very convenient. A way to deal with this is to first invent a coding into some other alphabet (e.g., {0, 1, #) and then recode this into binary. – Anyway, one natural way to encode $I = (A, w)$ would be to (i) write the sequence $\text{code}_0(I) = \text{bin}(w(a_1))\#...\#\text{bin}(w(a_n))$ of binary representations of the element weights, which gives an encoding over {0, 1, #}, and then (ii) recode $\text{code}_0(I)$ into {0, 1}, e.g., by putting $0 \rightarrow 00$, $1 \rightarrow 01$, $\# \rightarrow 10$.

Problem 5 (20 points). A problem P in a complexity class **C** is called *linear-time complete* for **C** iff for every problem $P' \in$ **C** there is a linear-time reduction from P' to P. Prove that there can be no linear-time complete problems for **P**. *Note: this problem was found to be identical to an old homework problem during the final exam and was to be ignored.*

Solution. Assume there is a linear-time complete problem $P \in$ **P**. Then $P \in \text{TIME}(p(n))$ for some polynomial p. By the time hierarchy theorem, there is a problem $P' \in \text{TIME}(p(n)^3) \subseteq$ **P** such that $P' \notin \text{TIME}(p(n))$. Using the linear-time reduction to P, we can construct a decision algorithm for P' which first reduces P' to P (in time $O(n)$), then applies the algorithm of P to the result. This yields an overall time consumption of at most $O(n) + p(kn) = O(n) + k'p(n) = O(p(n))$ for deciding P', which by linear speedup can also be achieved in time $p(n)$. This contradicts $P' \notin \text{TIME}(p(n))$, therefore the assumption that there is a linear-time complete problem $P \in$ **P** cannot hold.

Problem 6 (10 points). For a class **C** of languages (over some fixed alphabet), *co*-**C** denotes the class of the complement languages, that is, *co*-**C** = $\{L \mid L^c \in \mathbf{C}\}$. (a, 5 points) Show that **P** = *co*-**P**. (b, 5 points) Most researchers believe that **NP** ≠ *co*-**NP**. Explain why – if it is true – it is very likely very difficult to prove.

Solution. (a) If we have a deterministic TM deciding some language L in polynomial time $p(n)$, then by reversing the "yes" and "no" answers we get another TM deciding L^c in the same – that is, polynomial – time. Thus *co*-**P** ⊆ **P**. By symmetry we also have **P** ⊆ *co*-**P**.

(b) If we had proven NP ≠ ***co*-**NP, **then with (a) it would immediately follow that** P ≠ NP **– something which by all experience is hard to demonstrate.**

Computability and Complexity, Fall 2008: Final Exam - Solutions

***Revised** solution sheet, May 23, 2008*

Notes: A maximum of 100 points is accredited for this exam (sum of points from all problems is 115). Points of a problem are proportional to expected working time, not to expected difficulty. Best wishes!

Problem 1. (a, 10 points) Give a variant of the definition of nondeterministic single-tape TMs (Def. 9.1 in the lecture notes), which differs from the standard model in that the cursor, in addition to the standard moves, may be allowed to jump to an *arbitrary* cell on the tape. Furthermore, provide a definition of single-update transitions $(q, w, u) \xrightarrow{M} (q', w', u')$ for the new arbitrary-jump case, similar to the patterns in definition 3.3. (a) of the lecture notes. **(b, 10 points)** Provide an argument why these "jumper-TMs" cannot accept more languages than ordinary nondeterministic TMs.

Solution. (a) A nondeterministic (single-tape) jumper Turing machine is a structure $M = (K, \Sigma, \Delta, s)$, where K, Σ, s are defined like in deterministic TMs, just as we also keep our conventions concerning the special symbols $\sqcup$ and $\triangleright$. Δ is a relation

$$\Delta \subseteq (K \times \Sigma) \times [(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -, !\}]$$

If (q, w, u) and (q', w', u') are two configurations, we say that Δ takes M from (q, w, u) to $\xrightarrow{M} (q', w', u')$ in a single step, written $(q, w, u) \xrightarrow{M} (q', w', u')$, if one of the following cases holds:

(i) $w = va$ and $w' = va'$ and $u = u'$ and there is a rule of the form $((q, a), (q', a', -)) \in \Delta$
(ii) $w = va$ and $w' = v$ and $u' = a'u$ and there is a rule of the form $((q, a), (q', a', \leftarrow)) \in \Delta$
(iii) $w = va$ and $u' = bu''$ and $w' = va'b$ and $u' = u''$ and there is a rule of the form $((q, a), (q', a', \rightarrow)) \in \Delta$
(iv) $wu = vau$ and $w'u' = va'u$ and there is a rule of the form $((q, a), (q', a', !)) \in \Delta$

(only the last case was requested in the problem statement)

(b) Jumper-TMs cannot accept more languages because we can turn every jumper-TM into an equivalent normal nondeterministic TM by replacing every !-rule $((q, a), (q', a', !))$ with a rule $((q, a), (r_{q'}, a', !))$, using a novel state $r_{q'}$ not in K, and adding for all states $q \in K$ and $a \in \Sigma$ novel rules $((r_q, a), (r_q, a, \rightarrow))$, $((r_q, a), (r_q, a, \leftarrow))$ and $((r_q, a), (q, a, -))$.

Problem 2 (20 points). Give a lambda-expression **counts** that evaluates to the infinite list 0::1::2::3 … . In the makeup of **counts** you may use the list operators ::, **head**, **tail** and the integer function **succ**. You may also use, of course, the fixed point combinator Y.

Solution. We first procure a λ-expression for a function **add1** that takes a right-infinite list of integers and transforms it into the same list, but with each element incremented by 1. The naïve recursion for **add1** would be

add1 L = **succ** (**head** L) :: **add1** (**tail** L).

With the aid of Y, this turns into

> **add1** $\equiv$ Y(λgL. **succ** (**head** L) :: g(**tail** L)).

The naïve recursion for **counts** is **counts** = 0 :: **add1 counts**, with becomes the λ-expression Y(λg. 0 :: **add1** g). Inserting the λ-expression for **add1** this finally gives

> **counts** $\equiv$ Y(λg. 0 :: (Y(λhL. **succ** (**head** L) :: h(**tail** L))) g),

where for better readability and "variable-hygiene" we have replaced the g in the **add1** expression by h (although this is not necessary).

Problem 3 (15 points). Consider the language $H_d = \{<M> \mid Code(<M>)$ and M decides a language$\}$. Show that H_d is undecidable. You may use all results from section 6 in the lecture notes.

Solution. Notice that $H_d = \{<M> \mid Code(<M>)$ and M halts on every input with "yes" or "no"$\}$. The easiest way to prove the claim is to reduce $H_1 = \{<M> \mid Code(<M>)$ and M halts on all inputs$\}$ (known to be undecidable by Prop. 6.3) to H_d. For any TM N, one can effectively construct a TM N' which behaves exactly like N except that when N halts by going to the halting state h, N' halts with "yes". Then $N \in H_1$ iff $N' \in H_d$. Hence, if H_d were decidable, so would be H_1, which we know isn't, hence H_d is undecidable.

Problem 4 (10 points). Consider the following variant of Rice's theorem (derived from the original by replacing "accepting" by "deciding"):

> Let $L_C = \{<M> \mid Code(<M>)$ and M is a TM that decides a language $L \subseteq \Sigma^*$ which has property $C\}$.
>
> Now suppose that C is a nontrivial subset of the class of decidable languages over Σ. ("Nontrivial" means: not empty and not the class of decidable languages itself). Then L_C is undecidable.

Is this conjecture true or false? Give a proof of your answer.

Solution. The conjecture is true (in the solution sheet posted directly after the exam, I provided a faulty "proof" of the opposite claim... shame on me). The most direct way to see this is just to repeat the proof of the original Rice's theorem; it works for M deciding L exactly as for the original case of M accepting L.

A beautiful direct proof was found by Felix Schlesinger in the exam. It goes like this: Since C $\neq \emptyset$, if we could decide L_C we could find some M which decides a language $L \in C$. Now let N be an arbitrary TM. We can effectively modify N into N' such that for any input x, $N'(x)$ first simulates N on empty input, and if this halts, continues to simulate M on input x. Then N' will decide L iff N holds on the empty input, i.e. $N' \in L_C$ iff N holds on the empty input. This would imply that it were decidable whether some N holds on the empty input, which however we know (from a lecture notes proposition) is not decidable.

Problem 5 (20 points). Show that there exists a decidable language over $\{0, 1\}$ which is not in NP. Hint: one quick way to show this is by diagonalization.

Solution. Let (M_i, k_i) (i = 1, 2, ...) be an effective enumeration of all pairs of nondeterministic TMs M with tape alphabet {0, 1} and positive integers k. Let L_i be the language decided by M_i within time $|x|^{k_i}$. Notice that the set of all these languages is NP, and that every such L_i is decidable. Let $w_1, w_2, \ldots$ be the lexicographic enumeration of $\{0,1\}^*$. Then the diagonal language $\{w_i \in \{0,1\}^* \mid w_i \notin L_i\}$ is decidable and different from all of the L_i, hence not in NP.

Problem 6. Consider the following problem BIN PACKING:

BIN PACKING: Instance: a finite set U of items, a positive integer size $s(u)$ for every item, a positive integer bin capacity B, a number K of bins. Question: can the items be packed into the bins, that is, is there a partition of U into K subsets U_i ($i = 1, \ldots, K$) such that the sum of the sizes of items in each U_i is B or less?

(a, 5 points) Sketch a coding of problem instances using a coding alphabet $\Sigma = \{0, 1, \#\}$. **(b, 20 points)** Using your coding, show that BIN PACKING is in NTIME ($O(n^2)$).

Solution. (a) A straightforward coding of an instance would be, e.g., the string $x = bin(B)\#bin(K)\#bin(s(u_1))\#...\#\ bin(s(u_{|U|}))$, where $bin(x)$ is the binary representation of an integer x. **(b)** We design a nondeterministic multitape TM M which operates as follows. First, M writes a random string S of Σ^* of length at most $\log(K) \cdot |U|$ on its first working tape. Time needed: $O(\log(K) \cdot |U|) \leq O(\log(n)\ n)$, where $n = |x|$. Second, M checks whether S is of the form $bin(k_1)\#...\#bin(k_{|U|})$, where each $k_i \leq K$. If check fails, reject. Time needed: $O(|S|) \leq O(\log(n)\ n)$. Intended interpretation: k_i is the bin index where the i-th item goes. Third, for every $1 \leq k \leq K$, M adds all $s(u_i)$ where $k_i = k$. If any of these K sums exceeds B, reject, otherwise accept. Time needed for each of the sums plus check: $O(n^2)$. The overall time is thus $O(n^2)$.