Exercise Sheet Solutions 4

Compiler Design Lab
Prof. Dr. Sebastian Hack
Fabian Ritter, B.Sc.

This is a solution proposal. It does not necessarily describe complete solutions but gives the initial ideas to solve the theoretical exercises and presents the results.

Exercise 4.1 Properties of $\mathrm{LR}_0(G)$

Consider again the canonical LR(0) automaton that is constructed from the characteristic automaton of some context free grammar G via the subset construction (e.g. for the simple expression grammar as presented in the lecture). One can observe that for every state q of $\mathrm{LR}_0(G)$ that is not the initial state of the automaton, the incoming transitions of q all are under the same symbol.

In formulas, for $\mathrm{LR}_0(G) = (Q, \Sigma, \Delta, q_0, F)$:

$$\forall q, q_1, q_2 \in Q.\ \forall a, b \in \Sigma.\ (\Delta(q_1, a) = q \wedge \Delta(q_2, b) = q) \Rightarrow a = b$$

Explain why this property holds (you do not need to give a formal proof).

Solution: The subset construction can only produce an automaton that does not satisfy the above condition for an input automaton that contains an ε-strongly-connected-component with (at least) two incoming non-epsilon transitions under distinct symbols. (An ε-strongly-connected-component is a subset S of the states of the automaton where for each pair $s_1, s_2 \in S$ there is a path of ε-transitions from s_1 to s_2.)

No characteristic automaton of a context free grammar can contain such a structure:

- For characteristic automata, ε-strongly-connected-components with more than one state can only consist of states of the form $[A \to .B\omega]$. These do not have non-ε-predecessors.
- Every state subset that contains only one state is also an ε-strongly-connected-component. However, because of the structure of the characteristic automaton where each state corresponds to a position while processing a single production, each state has either exactly one incoming non-ε-transition or only incoming ε-transitions.

Therefore, the above property holds for every canonical LR(0) automaton.

Exercise 4.2 Type Inference

Consider the following program written in the toy functional language given in the lecture:

```
let rec f = fun ys → fun xs →
    if xs = [] then
        ys
    else
        ((f ((head xs) :: ys)) (tail xs))
in (f [])
```

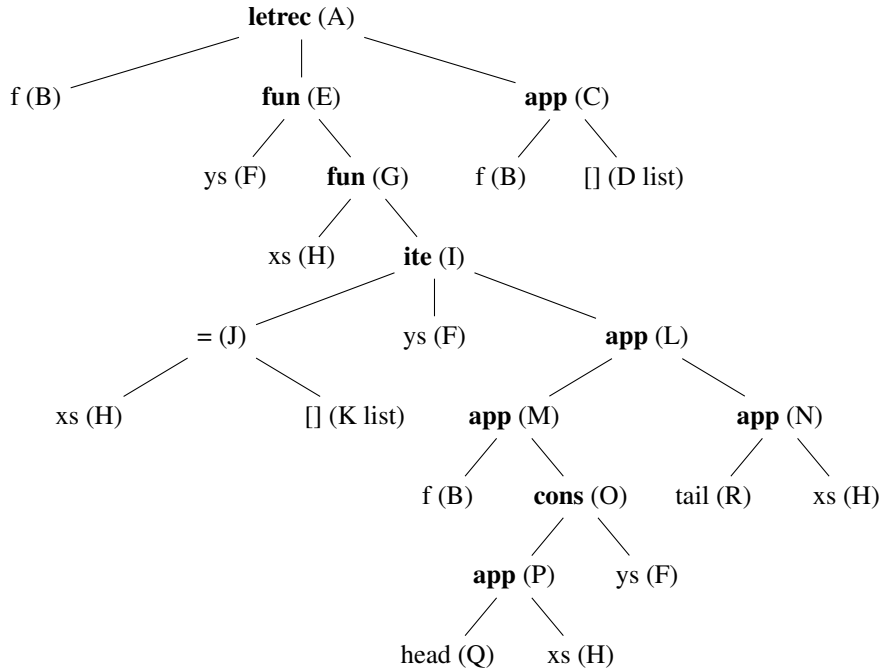
We extend the language with the following built-in functions operating on lists:

$$head : \alpha\ \mathrm{list} \to \alpha$$
$$tail : \alpha\ \mathrm{list} \to \alpha\ \mathrm{list}$$

1. Construct the abstract syntax tree for the program.
2. Type-check the program by constructing the set of equations and computing a most general unifier with the algorithms presented in the lecture. What is the type of the top-level expression?
3. Type-check the program with the syntax-driven algorithm $\mathcal{W}$ presented in the lecture.
4. What does the program compute?

Solution:

1. We already annotate names for the types of subexpressions in parentheses:



2. By applying the algorithm from the lecture, we construct the following system of equations:

$$
\begin{aligned}
B &= E & M &= N \to L \\
A &= C & B &= O \to M \\
E &= F \to G & F &= P \text{ list} \\
G &= H \to I & O &= P \text{ list} \\
J &= \textbf{bool} & Q &= H \to P \\
I &= F & Q &= \alpha \text{ list} \to \alpha && \text{(def. head)} \\
I &= L & R &= H \to N \\
H &= K \text{ list} & R &= \beta \text{ list} \to \beta \text{ list} && \text{(def. tail)} \\
J &= \textbf{bool} & B &= D \text{ list} \to C
\end{aligned}
$$

When applying the algorithm for the most general unifier, we add the following mappings to our unifier:

$B \mapsto E$	
$A \mapsto C$	
$E \mapsto F \to G$	
$G \mapsto H \to I$	
$J \mapsto \textbf{bool}$	
$I \mapsto F$	
$F \mapsto L$	
$H \mapsto K$ list	
$M \mapsto N \to L$	
$L \mapsto O$	from $L \to (K \text{ list} \to L) = O \to (N \to L)$
$N \mapsto K$ list	from $L \to (K \text{ list} \to L) = O \to (N \to L)$
$O \mapsto P$ list	
$Q \mapsto K$ list $\to P$	
$K \mapsto \alpha$	from $K \text{ list} \to P = \alpha \text{ list} \to \alpha$
$P \mapsto \alpha$	from $K \text{ list} \to P = \alpha \text{ list} \to \alpha$
$R \mapsto \alpha$ list $\to \alpha$ list	
$\alpha \mapsto \beta$	from $\alpha \text{ list} \to \alpha \text{ list} = \beta \text{ list} \to \beta \text{ list}$
$D \mapsto \beta$	from $\beta \text{ list} \to (\beta \text{ list} \to \beta \text{ list}) = D \text{ list} \to C$
$C \mapsto (\beta$ list $\to \beta$ list$)$	from $\beta \text{ list} \to (\beta \text{ list} \to \beta \text{ list}) = D \text{ list} \to C$

Overall, the mgu is:

$A \mapsto (\beta$ list $\to \beta$ list$)$	$K \mapsto \beta$
$B \mapsto \beta$ list $\to \beta$ list $\to \beta$ list	$L \mapsto \beta$ list
$C \mapsto (\beta$ list $\to \beta$ list$)$	$M \mapsto \beta$ list $\to \beta$ list
$D \mapsto \beta$	$N \mapsto \beta$ list
$E \mapsto \beta$ list $\to \beta$ list $\to \beta$ list	$O \mapsto \beta$ list
$F \mapsto \beta$ list	$P \mapsto \beta$
$G \mapsto \beta$ list $\to \beta$ list	$Q \mapsto \beta$ list $\to \beta$
$H \mapsto \beta$ list	$R \mapsto \beta$ list $\to \beta$ list
$I \mapsto \beta$ list	$\alpha \mapsto \beta$
$J \mapsto \textbf{bool}$	$\beta \mapsto \beta$

Therefore, the type A of the top-level expression is $(\beta$ list $\to \beta$ list$)$.

3. The algorithm $\mathcal{W}$ traverses the expression tree in a post-order fashion adding entries to the candidate unifier for the subexpressions that it visits. For this example, it adds the following non-trivial mappings (in the given order):

$$
\begin{aligned}
H &\mapsto K \text{ list} \\
J &\mapsto \textbf{bool} \\
Q &\mapsto \alpha \text{ list} \to \alpha \\
H &\mapsto \alpha \text{ list} \\
P &\mapsto \alpha \\
F &\mapsto \alpha \text{ list} \\
O &\mapsto \alpha \text{ list} \\
B &\mapsto \alpha \text{ list} \to M \\
R &\mapsto \beta \text{ list} \to \beta \text{ list} \\
\beta &\mapsto \alpha \\
N &\mapsto \alpha \text{ list} \\
M &\mapsto \alpha \text{ list} \to L \\
L &\mapsto \alpha \text{ list} \\
I &\mapsto \alpha \text{ list} \\
G &\mapsto \alpha \text{ list} \to \alpha \text{ list} \\
E &\mapsto \alpha \text{ list} \to (\alpha \text{ list} \to \alpha \text{ list}) \\
D &\mapsto \alpha \text{ list} \\
C &\mapsto \alpha \text{ list} \to \alpha \text{ list} \\
A &\mapsto \alpha \text{ list} \to \alpha \text{ list}
\end{aligned}
$$

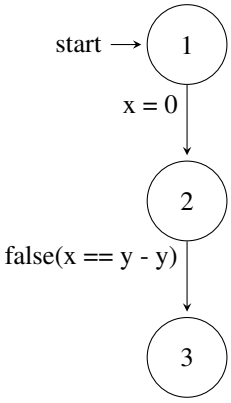
We can see that in the end, we can extract the same types (modulo renaming) from this mapping as we get from the previous part of the exercise.

4. Strictly speaking, this is a trick question: As we have not specified the dynamic semantics of the programming language, we cannot tell what an execution of it would mean. If we however follow the semantics known from e.g. OCaml and similar languages, the result of the program is a function that reverses arbitrary lists.

Exercise 4.3 Soundness in the Rules-of-Signs Analysis

Give a program (i.e. a control flow graph) and a corresponding computation of the fix-point of the Rules-of-Signs abstraction as presented in the lecture such that an intermediate result (which is not yet a fix-point) is already a sound approximation of the concrete program semantics.

Solution: Consider the following program (where x and y are parameters):



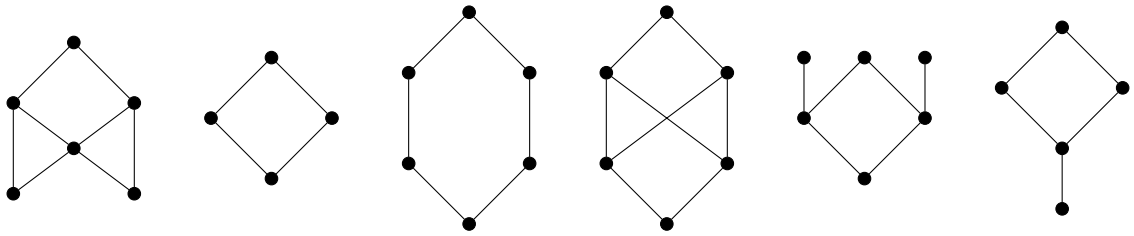
When applying the analysis, we get:

1		2		3	
x	y	x	y	x	y
$\top$	$\top$	$\bot$		$\bot$	
$\top$	$\top$	0	$\top$	$\bot$	
$\top$	$\top$	0	$\top$	0	$\top$

However, the second row is already a sound approximation of the possible concrete program behaviors as the condition to reach 3 can never be fulfilled.

Exercise 4.4 Finite Lattices

1. Which of the partially ordered sets described by the following Hasse diagrams are lattices? Which of them are complete lattices?



2. Relate the set of all finite lattices with the set of all finite complete lattices. Prove your claims.

Solution:

1. From left to right, the following classifications apply:
   - no lattice, no complete lattice (the lower two elements have no greatest lower bound)
   - lattice, complete lattice
   - lattice, complete lattice
   - no lattice, no complete lattice (the upper pair of elements on the same horizontal layer has no unique greatest lower bound)
   - no lattice, no complete lattice (no two of the maximal elements have a least upper bound)
   - lattice, complete lattice

2. The set of all finite complete lattices is a strict subset of the set of all finite lattices.

   Proof. Each complete lattice is also a lattice: Let $(L, \sqsubseteq)$ be a complete lattice. We define pairwise join $\sqcup$ and meet $\sqcap$ as follows:

   $$a \sqcup b = \bigsqcup\{a, b\}$$
   $$a \sqcap b = \bigsqcap\{a, b\}$$

   From the completeness of $(L, \sqsubseteq)$, we know that the set-wise join and meet operations are well defined and satisfy the requirements for the pair-wise join and meet operations.

   For the strictness of this relation, consider the empty set $\varnothing$ (ordered by the empty relation): For all pairs (a, b) of elements in $\varnothing$, *false* holds, therefore also $\sup(a, b) \in \varnothing$ and $\inf(a, b) \in \varnothing$. Thus, $\varnothing$ is a lattice. However, $\varnothing \subseteq \varnothing$ holds, therefore $\varnothing$ cannot be a complete lattice, as it does not contain a greatest lower bound for $\varnothing$.