CS2PLD: Programming Language Design Paradigms

Exam May 2022

Question 1:

1. Consider the program P below in an imperative language where variables can store numbers (type Int) or Booleans (type Bool), the command `read(y)` is used to input a value and store it in the variable `y`, the syntax `:=` denotes assignment, + denotes addition and = equality.

```
program main
var x : Bool
var y : Int
begin main
read(y);
x := True;
if y = 0 then x := x+1;
end main
```

Now answer the following questions, providing explanations for your answers (answers without explanation will carry 0 marks).

(a) Assuming the language has a static type system, will the program P given above be accepted?

(b) Assuming the language has a dynamic type system, will the program P given above be accepted?

—

A static type checker will reject the example program because x is used once as Bool and once as Int.

A dynamic type checker could accept the program and raise an error if the value read for y is zero.

2. Solidity is a programming language used to write smart contracts for the Ethereum blockchain. The following grammar defines the abstract syntax for a simplified version of Solidity, where id, n, b and a denote tokens representing an identifier, a number, a boolean and an address, respectively.

$$
\begin{array}{lll}
S & ::= & C \mid C \cdot S \\
C & ::= & \texttt{contract}\ id\ \{StList\} \\
StList & ::= & St \mid St\ ;\ StList \\
St & ::= & Method \mid StateDef \\
StateDef & ::= & Type\ id \mid Type\ id\ =\ E \\
Type & ::= & \texttt{uint} \mid \texttt{bool} \mid \texttt{addresss} \mid \texttt{address payable} \\
Method & ::= & Funct \mid Stmt \\
Funct & ::= & ... \\
Stmt & ::= & (Stmt\ ;\ Stmt) \mid \\
 & & \texttt{if}\ (E)\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt \mid \\
 & & \texttt{while}\ (E)\ \texttt{do}\ Stmt \mid \\
 & & id = E \mid \\
 & & E.\texttt{transfer}(E) \mid \texttt{skip} \mid \ldots \\
E & ::= & id \mid n \mid b \mid a \mid E\ Op\ E \mid \texttt{msg.sender} \mid \texttt{msg.value} \\
Op & ::= & + \mid - \mid * \mid / \mid > \mid < \mid = \mid \texttt{or} \mid \texttt{and}
\end{array}
$$

(a) List three terminal and two non-terminal symbols in this grammar.

—

Terminals: id, (,), `contract`, =, +, etc
Non-terminals: S, C, $StList$, St, $StateDef$, $Type$ etc.

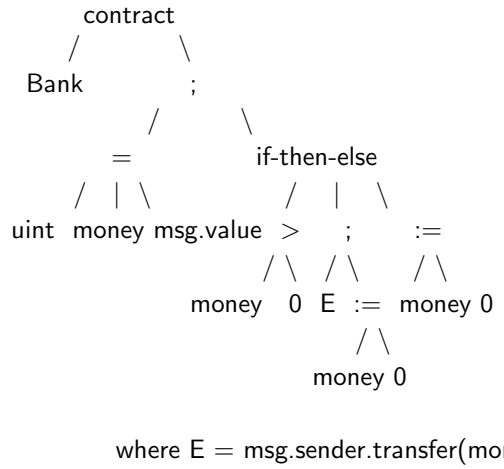
(b) Consider the following programs. In each case, either draw an abstract syntax tree to represent the program (if it is derivable from the grammar given above), or indicate why the program is not syntactically correct.

i. `contract` $Bank\{$
`uint` $money =$ `msg.value`;
`if` $(money > 0)$
`then` (`msg.sender.transfer`$(money)$; $money = 0$)
`else` $money = 0\}$

ii. `contract` $Bank$ $\{$
`bool` $money =$ `msg.value`;
`if` $(money > 0)$
`then` (`msg.sender.transfer`$(money)$; $money = 0)\}$

—

The first program is represented by an abstract syntax tree with root labelled contract and two subtrees: one with the name (Bank), the other representing the list of statements (consisting of a State Def and a command).



Note that the State subtree has root labelled = and subtrees uint, money, msg.value. The other subtree also has only one component: the root is labelled if-then-else and there are three subtrees (representing the condition money $>$ 0, the "then" branch, which is a sequence of commands `msg.sender.transfer`(money); money = 0, and the "else" branch, which is the assignment money = 0).

The second program is not syntactically correct, because it is missing the else. It cannot be derived from the grammar. Note that it also has a wrong type for the variable money, but this is not the reason why the program is not derivable.

3. Give a principle of induction that could be used to prove that a certain property P holds for all the expressions in Solidity, i.e., a principle of induction to prove $\forall E.P(E)$ where E is an expression as defined by the grammar above.

—

To prove $\forall E.P(E)$ where E is an expression as defined above, we need to prove:

Base cases: $P(id), P(n), P(b), P(a), P(msg.sender), P(msg.value)$ for all id, n, a, b.

Induction Step: If $P(E_1)$ and $P(E_2)$ hold then $P(E_1\ Op\ E_2)$ holds for any Op.

4. Solidity has two kinds of memory, a local memory in the Ethereum Virtual Machine, and the blockchain memory. To define the operational semantics of the language, we use configurations of the form $\langle c, \sigma\rangle$ where c is a program (or a program component) and σ is a pair consisting of the local memory ρ and the blockchain storage μ. Both ρ and μ are partial functions from identifiers to values (similar to the way the memory is represented in SIMP).

   The semantics of the statement $id = E$ is defined by the following rule (big-step semantics). Explain in your own words the behaviour of this command.

$$\frac{\langle E, (\rho, \mu)\rangle \Downarrow \langle v, (\rho, \mu)\rangle}{\langle id = E, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle}$$

$$\text{where if } id \in dom(\rho) \text{ then } \rho' = \rho[id \mapsto v] \text{ and } \mu' = \mu,$$
$$\text{otherwise } \rho' = \rho,\ \mu' = \mu[id \mapsto v].$$

—

It is an assignment. If id exists in the local memory ρ, then the value of E is stored locally at id, otherwise, it is stored in the blockchain memory μ (i.e., priority is given to the local storage).

5. Contracts are executed sequentially (left to right), that is, to evaluate a program $c_1 \cdot c_2$ consisting of two contracts, first the contract c_1 is evaluated and then c_2 is evaluated. Initially the local memory is empty (nil), that is, the first contract is executed in the state (nil, μ). Each contract is evaluated on the state returned from the execution of the previous one, replacing however the local memory ρ with a new empty memory each time, that is, the local memory is not preserved from one contract to the next, only the blockchain memory is preserved.

   Give a formal definition of the semantics of a program of the form $C \cdot S$ (see the grammar in part (2): C is a contract and S one or more contracts). You can give a big-step or a small-step semantics.

—

$$\frac{\langle C, (nil, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle \quad \langle S, (nil, \mu')\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle}{\langle C \cdot S, (nil, \mu)\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle}$$

Question 2:

1. Recall that `++` denotes list concatenation and that the notation `x:xs` represents the list with head `x` and tail `xs`.

   Now consider the functions `myseq` and `comp` below:

```
myseq x d 0 = []
myseq x d 1 = [x]
myseq x d n
   | n > 1 = (myseq x d (n-1))++[x+d*(n-1)]
   | n < 0 = error "Invalid sequence length"

comp [] = 0
comp (x:xs) = x + comp(xs)
```

   Answer the questions below:

   (a) State in plain English what the functions `myseq x d n` and `comp (x:xs)` compute. Illustrate your answer by computing the values of `myseq 1 1 3` and `comp [1,2,3]`. Give all intermediate values in the computation.

   —

   As we can see from the definition of the function `myseq`, the function produces a list of n elements, starting with the element x, and with difference d between adjacent elements. In other words, it produces the arithmetic sequence with `n` elements, whose first term is `x` and with common difference `d`. The function `comp xs` computes the sum of the numbers in the list `xs`.

   For example

   `myseq 1 1 3`=(myseq 1 1 2)++[3]=[1,2,3]
   `myseq 1 1 2`=(myseq 1 1 1)++[2]=[1,2]
   `myseq 1 1 1`=[1]
   `myseq 1 1 0`=[]

   `comp [1,2,3]=1+(comp [2,3])=1+2+(comp [3])=1+2+3+(comp [])=1+2+3+0=6`

   (b) Prove by induction that for all natural numbers `n`:

$$\texttt{comp (myseq x d n)} = \frac{n(2x + d(n-1))}{2}.$$

   Hint: You can use the fact that `comp (xs ++ ys) = (comp xs) + (comp ys)`

   —

   (Base case) For $n = 0$, we have that `comp (myseq x d 0)=comp []` $= 0$. Indeed,

$$\begin{aligned} \frac{n(2x + d(n-1))}{2} &= \frac{0(2x - d)}{2} \\ &= 0 \end{aligned}$$

   (IH) Now assume that for a given n, `comp (myseq x d n)` $= \frac{n(2x + d(n-1))}{2}$.

   We want to show that

$$\texttt{comp (myseq x d (n+1))} = \frac{(n+1)(2x + dn)}{2}$$

By the definition of `myseq`,

$$\texttt{myseq x d (n+1)=(myseq x d n)++[x+nd].}$$

Given the hint,
`comp (myseq x d n)++[x+nd]=comp (myseq x d n)+comp [x+nd]`. By the IH,

$$\texttt{comp (myseq x d n)} = \frac{n(2x+d(n-1))}{2},$$

hence

$$\begin{aligned}
\texttt{comp (myseq x d n)++[x+dn]} &= \frac{n(2x+d(n-1))}{2} + \texttt{comp [x+nd]} \\
&= \frac{n(2x+d(n-1))}{2} + (x+dn) \\
&= \frac{n(2x+d(n-1)) + 2x + 2dn}{2} \\
&= \frac{2xn + dn(n-1) + 2x + 2dn}{2} \\
&= \frac{2x(n+1) + dn(n-1+2)}{2} \\
&= \frac{2x(n+1) + dn(n+1)}{2} \\
&= \frac{(n+1)(2x+dn)}{2}
\end{aligned}$$

2. Consider the two functions `f` and `g` below.
```
f x xs = (x:xs)
g (x:xs) = x
```
Answer the questions below.

   (a) Give polymorphic types for the functions `f` and `g`.

   —
   ```
   f ::  a -> [a] -> [a].
   g ::  [a] -> a.
   ```

   (b) For each of `(f.g)` and `(g.f)`, if the composition is well-defined, give its type. If the composition is not well-defined, explain why this is the case. You do not have to provide the type derivations, but must explain how you reached your conclusion.

   —
   The composition `f.g` is well-defined and has type `(f.g) ::  [a] -> [a] -> [a]` (this comes straight from the application of the definition of composition).
   The composition `(g.f)` is not well-defined, because the domain of `g` is `[a]` but the co-domain of `f` is `[a]->[a]`.

3. Consider the SFUN program below.

```
double(x) = 2 * x
```

```
square(x) = x * x
min(x,y) = if x < y then x else y
reward(x,y) = if x=y then double(x) else min(x,y)
```

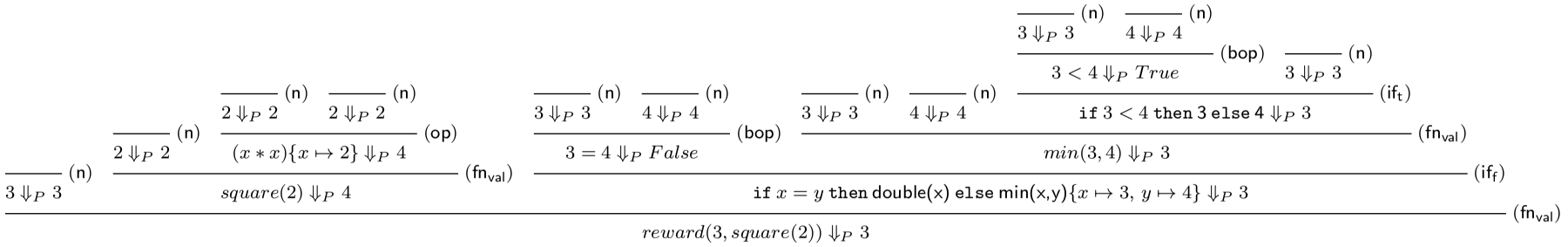
Show the evaluation of the term

```
reward(3,square(2))
```

using the Big-Step Semantics with the *call-by-value* strategy (use the relation $\Downarrow_P$).

—

Using the Big-Step Semantics:


3 ⇓P 3 (n)
2 ⇓P 2 (n)
2 ⇓P 2 (n)
(x ∗ x){x ↦ 2} ⇓P 4 (op)
square(2) ⇓P 4 (fnval)
3 ⇓P 3 (n)
4 ⇓P 4 (n)
3 = 4 ⇓P False (bop)
3 ⇓P 3 (n)
4 ⇓P 4 (n)
3 ⇓P 3 (n)
4 ⇓P 4 (n)
3 < 4 ⇓P True (bop)
3 ⇓P 3 (n)
if 3 < 4 then 3 else 4 ⇓P 3 (ift)
min(3, 4) ⇓P 3 (fnval)
if x = y then double(x) else min(x,y){x ↦ 3, y ↦ 4} ⇓P 3 (iff)
reward(3, square(2)) ⇓P 3 (fnval)

4. Determine whether the unification problem

$$\mathcal{U}=\{\texttt{p([H|T])=p([1,2,3]),q(f(A),X,T)=q(f(Y),g(H),[Y|Z])}\}$$

is solvable. If $\mathcal{U}$ is solvable, give its most general unifier. Otherwise, explain why $\mathcal{U}$ is not solvable.

—

```
p([H|T])=p(1,2,3) => (H=1; T=[2,3])
q(f(A),X,T)(H/1; T/[2,3])=q(f(A),X,[2,3])
q(f(Y),g(H),[Y|Z])(H/1; T/[2,3])=q(f(Y),g(1),[Y|Z])
q(f(A),X,[2,3])=q(f(Y),g(1),[Y|Z]) =>
   (Y=A=2; X=g(1); Y=2 ;Z=[3])
```

mgu $\sigma = \{\texttt{H} = 1; \texttt{T} = [2,3]; \texttt{Y} = 2; \texttt{A} = 2; \texttt{X} = \texttt{g}(1); \texttt{Z} = [3]\}$.

5. For each of the following predicate logic formulas, if the formula can be converted into one or more Horn clauses, then give the clause(s) in Prolog notation and indicate what type of clause(s) they are. If the formula cannot be converted, explain why.

   (a) $\forall x(\neg p(x) \vee \neg q(x) \vee (r(x) \wedge s(x)))$

   (b) $\forall x((p(x) \wedge q(x)) \rightarrow (r(x) \vee s(x))$

   (c) $\forall x(\neg p(x) \vee \neg q(x) \vee \neg r(x))$

—

   (a) $\forall x(\neg p(x) \vee \neg q(x) \vee (r(x) \wedge s(x)))$:
   The rules
   `r(x) :- p(x), q(x).`
   and
   `s(x):- p(x), q(x).`

   (b) $\forall x((p(x) \wedge q(x)) \rightarrow (r(x) \vee s(x))$
   Cannot be converted because the resulting clause contains two positive literals and therefore it is not a Horn clause.

   (c) $\forall x(\neg p(x) \vee \neg q(x) \vee \neg r(x))$
   The goal
   `:- p(x), q(x), r(x).`

6. Consider the following Prolog program P:

```
desc(X,Y) :- parent(Y,X).
desc(X,Z) :- parent(Y,X), desc(Y,Z).
parent(paul,jane).
parent(jane,elliot).
```

Using the selection function that always selects the left-most literal in the goal, give *all* answers to the goal

```
:- desc(X,Y)
```

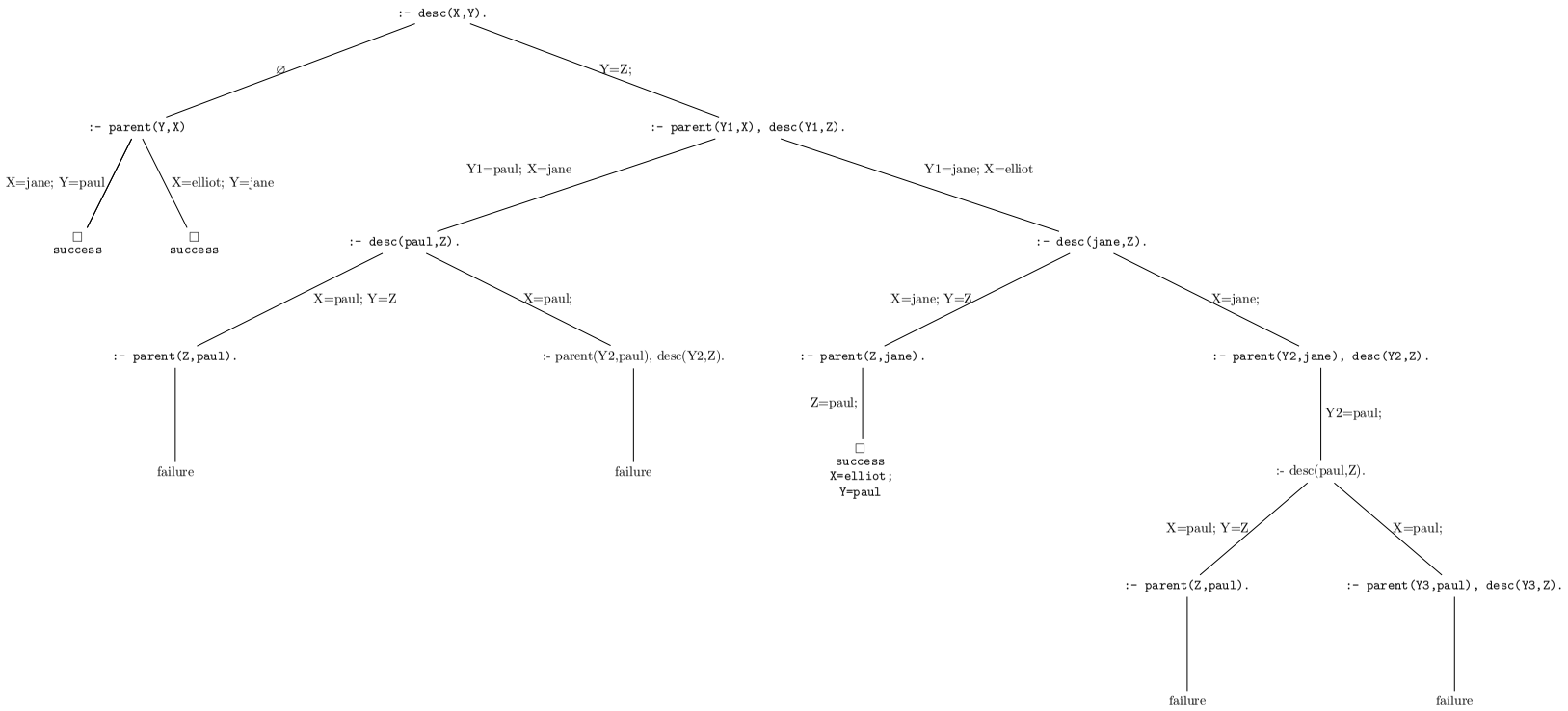
and draw the corresponding SLD-resolution tree. Indicate all variable bindings in the tree.

—

The goal produces answers:

```
X=jane; Y=paul
X=elliot; Y=jane
X=elliot; Y=paul
```

The SLD-Resolution tree is given next.


:- desc(X,Y).
∅
Y=Z;
:- parent(Y,X)
:- parent(Y1,X), desc(Y1,Z).
X=jane; Y=paul
X=elliot; Y=jane
Y1=paul; X=jane
Y1=jane; X=elliot
□
success
□
success
:- desc(paul,Z).
:- desc(jane,Z).
X=paul; Y=Z
X=paul;
X=jane; Y=Z
X=jane;
:- parent(Z,paul).
:- parent(Y2,paul), desc(Y2,Z).
:- parent(Z,jane).
:- parent(Y2,jane), desc(Y2,Z).
Z=paul;
Y2=paul;
failure
failure
□
success
X=elliot;
Y=paul
:- desc(paul,Z).
X=paul; Y=Z
X=paul;
:- parent(Z,paul).
:- parent(Y3,paul), desc(Y3,Z).
failure
failure