Programming Language Theory

Preparation for Exam

Ralf Lämmel
• Big-step semantics
• Small-step semantics
• Semantics-based reasoning
• Type systems
• The untyped lambda calculus
• The simply-typed lambda calculus
• Lambda calculi with polymorphism
• Featherweight Java
• Concurrency calculi
• Denotational semantics
• Program analysis
• (Program specialization)
Principles underling the exam

• The exam asks for tiny Prolog or Haskell or Greek snippets.

• The exam does not contain Multiple Choice.

• The exam does not ask for prose (see “riddles” though).

• Precision of notation is not important.

• Key notions from lectures must be mastered.

• Subjects/skills from assignments must be mastered.
Categories of questions for **midterm**
(10 questions in total, 1 per category except for 2 riddles.)

1. Define the **abstract syntax** of given constructs.

2. Define a **natural semantics** of a given language.

3. Define an **SOS semantics** of a given language.

4. Define a **type system** of a given language.

5. Draw a **derivation tree** for a given term.

6. Define a **denotational semantics** of a given language.

7. Define a **program analysis** for a given problem and language.

8. Demonstrate **declarative programming** skills.

9. Solve a **PLT riddle** with a succinct argument.
Languages appearing in the exam

- While
- B/NB
- Lambda calculi
- Featherweight Java
- CCS
- Tiny subsets of Java, Haskell, Prolog
- Popular data types (strings, sets, lists, stacks, ...)
- Domain-specific languages explained well
On the matter of notation

• Some task may require the use of Prolog or Haskell.
• Other tasks can be done in either Greek, Prolog, or Haskell.
• Minor notational issues do not affect grade.
• Any convention encountered in the course can be used.
Grading rules

- 0-2 points per question
  - 0 “missing or mental assault”
  - 1 “the beginning of an idea”
  - 2 “nearly or fully complete/correct”

- 2 possible extra points per exam
  - for up-to 2 “outstanding solutions”

- 20 points in total + 2 extra points
Samples questions and answers
“Define the abstract syntax of given constructs.”

It could be required to use Prolog or Haskell.
“Define the abstract syntax of given constructs in Prolog.”

The following domains describe the syntax of System F. **It’s enough to give Prolog clauses for category $t$.**

$$\begin{align*}
t &::= x | v | t \; t | t[T] \\
v &::= \lambda x : \; T. \; t | \land X. \; t \\
T &::= X | T \rightarrow T | \forall X. \; T
\end{align*}$$

One needs to know the “names” of the constructs.
Solution

isterm(var(X)) :- isvar(X).
isterm(V) :- isvalue(V).
isterm(app(T1,T2)) :- isterm(T1), isterm(T2).
isterm(tapp(T,Ty)) :- isterm(T), istype(Ty).
“Define the abstract syntax of given constructs in Prolog.”

Recall the essential operators of CCS, and devise a term-based Prolog representation. To this end, define a Prolog predicate term/1 whose extension is the set of valid CCS agents. Use good names for the term constructors (or add comments) so that it is clear what constructs you are talking about. You can leave out restriction, relabeling, and definitions of agent constants. You may also take for granted a predicate action/1 for actions.
Solution

\[
\text{term(seq(A,T)) :- action(A), term(T).} \quad \% \text{sequential combinator}
\]

\[
\text{term(T1+T2) :- term(T1), term(T2).} \quad \% \text{summation}
\]

\[
\text{tem(T1|T2) :- term(T1), term(T2).} \quad \% \text{composition}
\]
“Define the abstract syntax of given constructs in Prolog.”

Imagine a language for stack-based addition of integers. In some concrete syntax, a program could look like as follows:

```
push 42
push 42
add
```

(The result should be 84 for what it matters.) Devise an abstract syntax for operations and sequences.
One needs to observe informal elements (such as sequences of operations) in defining the syntax.
Non-optimal solution

op(push(X)) :- number(X).
op(add).
op(append(O1,O2)) :- op(O1), op(O2).

This approach would enable nested grouping while the intention is to limit the representation to flat sequences of ops. Nevertheless, this would still be considered a “good solution”.
Prolog was exercised above, but Haskell could be required.
“Define a natural semantics of a given language.”

It could be required to use Prolog or Haskell.
“Define a natural semantics of a given language in Prolog.”

Normal forms are Booleans and numbers.

```
term(num(N)) :- number(N).
term(add(T1,T2)) :- term(T1), term(T2).
term(iszero(T)) :- term(N).
term(cond(T0,T1,T2)) :- term(T0), term(T1), term(T2).
```
Solution

eval(num(N),N).
eval(add(T1,T2),N) :- eval(T1,N1), eval(T2,N2), N is N1 + N2.
eval(iszero(T),true) :- eval(T,0).
eval(iszero(T),false) :- eval(T,N), \(+ N == 0.
eval(cond(T0,T1,_),N) :- eval(T0,true), eval(T1,N).
eval(cond(T0,_,T2),N) :- eval(T0, false), eval(T2,N).
“Define a natural semantics of a given language in Prolog.”

Interpret terms for set expressions:

term(singleton(X)) :- integer(X).
term(union(T1,T2)) :- term(T1), term(T2).
term(intersection(T1,T2)) :- term(T1), term(T2).

The interpreter may assume helper predicates for union/2 and intersection/2.
ensure_loaded(library(lists)).
eval(singleton(X),[X]).
eval(union(T1,T2),R) :-
    eval(T1,R1), eval(T2,R2), union(R1, R2, R).
eval(intersection(T1,T2),R) :-
    eval(T1,R1), eval(T2,R2), intersection(R1, R2, R).

You don’t need to know / mention that part!
“Define a natural semantics of a given language.”

Consider terms such as \( z, s(z), s(s(z)) \), etc. Further, we assume that variables may occur in terms (read-access only). You can assume a suitable lookup function.
Solution

As it happens, this semantics is compositional.

evaluate(M,z,z).
evaluate(M,s(X),s(Y)) :- evaluate(M,X,Y).
evaluate(M,v(N),V) :- lookup(M,N,V).
Define a natural semantics of a given language.

(You are encouraged to use Prolog to represent the deduction rules in question.) Consider a trivial imperative, expression-oriented language with the following expression forms: 0 ("z"), successor ("s(...)"), assignment ("...=..."), variable reference ("v(...)"), and sequential composition ("(......)"). Here are some examples of expressions and their associated values:

\[
\begin{align*}
  s(s(z)) & \text{ evaluates to } 2 \\
  x = s(s(z)) & \text{ evaluates to } 2 \\
  (x = s(s(z)), s(v(x))) & \text{ evaluates to } 3
\end{align*}
\]

Define expression evaluation.
Hint: you need a memory for variables; the following predicates can be assumed.

\[
\begin{align*}
  \text{lookup} & (M,X,Y) :- \text{append}(_,[ (X,Y) | _ ], M). \\
  \text{update} & ([],X,Y,[(X,Y)]). \\
  \text{update} & ([ (X,_)|M ],X,Y,[(X,Y)|M]). \\
  \text{update} & ([ (X1,Y1)|M1 ],X2,Y2,[(X1,Y1)|M2]) :- \\
  \quad \text{\+} X1 = X2, \\
  \quad \text{update} (M1,X2,Y2,M2).
\end{align*}
\]
Solution

eval(z,M,0,M).
eval(s(T),M1,V2,M2) :- eval(T,M1,V1,M2), V2 is V1 + 1.
eval(v(X),M,V,M) :- lookup(M,X,V).
eval(X=T,M1,V,M3) :- eval(T,M1,V,M2), update(M2,X,V,M3).
eval((T1,T2),M1,V,M3) :- eval(T1,M1,_,M2), eval(T2,M2,V,M3).

% You can use library functionality.

:- ensure_loaded('map.pro').

% A demo (not required by a solution)

main
  :-
      eval(s(s(z)),[],V1,_), write(V1), nl, % prints 2
      eval(x=s(s(z)),[],V2,_), write(V2), nl, % ditto
      eval((x=s(s(z)),s(v(x))),[],V3,_), write(V3), nl. % prints 3
“Define an SOS semantics of a given language.”

It could be required to use Prolog or Haskell.
“Define an SOS semantics of a given language in Prolog.”

Consider terms such as 42, add(42,88), add(add(2,42),44), etc.

Hint: you need to come up with an extra relation for values (“normal forms”) to be able to adhere to small-step style.
Solution

step(add(X,Y),add(Z,Y)) :- step(X,Z).
step(add(X,Y),add(X,Z)) :- value(X),step(Y,Z).
step(add(X,Y),Z) :- value(X),value(Y), Z is X + Y.
value(X) :- number(X).
Alternative solution

step(add(X,Y),add(Z,Y)) :- step(X,Z).
step(add(num(X),Y),add(num(X),Z)) :- step(Y,Z).
step(add(num(X),num(Y)),num(Z)) :- Z is X + Y.
Consider a trivial programming language \textit{Hyphen} which can essentially print any number of hyphens. This language has the following constructs: \textit{skip} (i.e., the empty program), sequential composition (possibly denoted by “(...,...)”), \textit{hyphen} (to “print” a hyphen, i.e., to add a hyphen to a list of output values), a restricted form of loops to iterate a statement a given number of times (possibly denoted by “\textit{ntimes}(N,...)”). Here is an illustrative execution in Prolog:

\[
\text{?- manysteps(ntimes(7,hyphen),[],Output).}
\]
\[
\text{Output} = [-, -, -, -, -, -, -].
\]

Devise the step/4 relation for \textit{Hyphen}.
onestep(hyphen, skip, O1, O2) :- append(O1, ['-', '], O2).
onestep((skip, T), T, O, O).
onestep((T1, T2), (T3, T2), O1, O2) :- onestep(T1, T3, O1, O2).
onestep(ntimes(1, T), T, O, O).
onestep(ntimes(N1, T), (T, ntimes(N2, T)), O, O) :-
    N1 > 1,
    N2 is N1 - 1.

% star closure (not required by a solution)
manysteps(T1, O1, O3) :-
onestep(T1, T2, O1, O2) ->
    manysteps(T2, O2, O3)
; O3 = O1.

% A demo (not required by a solution)
main :-
    manysteps(ntimes(7, hyphen), [], O),
    write(O), nl.
“Define a type system of a given language.”

It could be required to use Prolog or Haskell.
“Define a type system of a given language in Prolog.”

You have ints and floats (consider them different forms of terms). Addition can be applied to either two ints or two floats.
Solution

Solution would be good enough w/o primitive type tests.

Solution would be outstanding if the last 2 rules were stated as 1.

```prolog
typeof(int(X), inttype) :- integer(X).
typeof(float(X), floattype) :- float(X).
typeof(add(X, Y), inttype) :- typeof(X, inttype), typeof(Y, inttype).
typeof(add(X, Y), floattype) :- typeof(X, floattype), typeof(Y, floattype).
```
Possibly outstanding solution

```prolog
typeof(int(X),inttype) :- integer(X).
typeof(float(X),floattype) :- float(X).
typeof(add(X,Y),T) :- typeof(X,T), typeof(Y,T).
```
“Define a type system of a given language.”

Consider an overloaded addition for types `int`, `float` and `string`. There are maybe other types in the language for which addition is not defined, e.g., `char`. Addition for number types (i.e., `int` and `float`) should also be overloaded for mixed operand types, in which case the type of addition should be `float`. Define all typing rules for addition.
typeOf(plus(T1,T2),string) :- typeOf(T1,string), typeOf(T2,string).
typeOf(plus(T1,T2),int) :- typeOf(T1,int), typeOf(T2,int).
typeOf(plus(T1,T2),float) :- typeOf(T1,float), typeOf(T2,float).
typeOf(plus(T1,T2),float) :- typeOf(T1,int), typeOf(T2,float).
typeOf(plus(T1,T2),float) :- typeOf(T1,float), typeOf(T2,int).
Outstanding solution

```prolog
typeOf(plus(T1,T2),T) :- plusable(T), typeOf(T1, T), typeOf(T2, T).

typeOf(plus(T1,T2),float) :- typeOf(T1,int), typeOf(T2,float).

typeOf(plus(T1,T2),float) :- typeOf(T1,float), typeOf(T2,int).

plusable(int).

plusable(float).

plusable(string).
```
“Draw a derivation tree for a given term.”

Derivation may be concerned with semantics or types. The derivation rules are typically given explicitly. (They may also be assumed as known: e.g., basic rules for \(\lambda\)/Haskell.)
“Draw a derivation tree for a given term.”

Typing rules

true : Bool
false : Bool
0 : Nat
x : Nat
s(x) : Nat
x : Nat, y : Nat
x + y : Nat

Term

0 + s(s(0))
Solution

\[
\begin{array}{c}
0 : \text{Nat} \\
\hline
s(0) : \text{Nat} \\
\hline
0 : \text{Nat} \\
\hline
s(s(0)) : \text{Nat} \\
\hline
0 + s(s(0)) : \text{Nat}
\end{array}
\]
Alternative solution
(Derivation trees = proof trees)

• Make assumptions for clarity (optional):
  ★ Assume a Prolog predicate typeof.
  ★ Use prefix terms for all constructs (e.g., add).
• Represent proof tree “by indentation”.
  ★ typeof(add(0, s(s(0))), nat)
    ★ typeof(0, nat)
    ★ typeof(s(s(0)), nat)
    ★ typeof(s(0), nat)
    ★ typeof(0, nat)
“Draw a derivation tree for a given term.”

Consider the following typing rules of the $NB$ language:

\begin{verbatim}
welltyped(true, bool).
welltyped(false, bool).
welltyped(zero, nat).
welltyped(succ(T), nat) :- welltyped(T, nat).
welltyped(pred(T), nat) :- welltyped(T, nat).
welltyped(iszero(T), bool) :- welltyped(T, nat).
welltyped(if(T1, T2, T3), T) :-
    welltyped(T1, bool),
    welltyped(T2, T),
    welltyped(T3, T).
\end{verbatim}

Give a typing derivation for the following term:

\[ \text{succ}(\text{if}(\text{iszero}(\text{zero}), \text{zero}, \text{succ}(<zero>))) \]
Solution

(Other representations of the derivation tree are also Ok.)

- wellTyped(succ(if(iszero(zero), zero, succ(zero))), nat)
  - wellTyped(if(iszero(zero), zero, succ(zero)), nat)
    * wellTyped(iszero(zero), bool)
      - wellTyped(zero, nat)
    * wellTyped(zero, nat)
    * wellTyped(succ(zero), nat)
      - wellTyped(zero, nat)
“Draw a derivation tree for a given term.”

Infer the Haskell type of the following expression:

(head, tail)
Solution

\[
\text{head} :: [a] -> a \quad \text{tail} :: [b] -> [b]
\]

\[
\text{(head, tail)} :: ([a] -> a, [b] -> [b])
\]
“Draw a derivation tree for a given term.”

Infer the Haskell type of the following expression:

```
head . filter fst
```
Solution

head :: [a] -> a

filter :: (b -> Bool) -> [b] -> [b]

fst :: (c,d) -> c

(.) :: (g -> f) -> (e -> g) -> e -> f

(.) head :: (e -> [a]) -> e -> a

  because g -> f = [a] -> a and hence g = [a], f = a

filter fst :: [(Bool,d)] -> [(Bool,d)]

  because (b -> Bool) = (c,d) -> c and hence c = Bool and b = (Bool,d)

(.) head (filter fst) :: [(Bool,d)] -> (Bool,d)

  because (e -> [a]) = [(Bool,d)] -> [(Bool,d)]

  and hence e = [(Bool,d)] and a = (Bool,d)
“Define a denotational semantics of a given language.”
“Define a denotational semantics of a given language.”

Consider the following abstract syntax of a simple state machine (think of Java byte code):

\[
\text{data Code} = \text{Push Int} \quad -- \text{push an element onto the stack} \\
| \text{Add} \quad -- \text{replace topmost elements by sum} \\
| \text{Seq Code Code} \quad -- \text{left-to-right composition} \\
| \text{While2 Code} \quad -- \text{loop until stack size < 2}
\]

Define a denotational semantics so that the following main program would print a stack with the single element 10. Stacks are represented as lists.

```haskell
main = 
do
  print $ exec (Seq (Push 1))
      (Seq (Push 2))
      (Seq (Push 3))
      (Seq (Push 4))
      (While2 Add))))) []
```
Solution

exec :: Code -> [Int] -> [Int]
exec (Push i) = (i:)
exec Add = \(i1:i2:s\) -> (i1+i2:s)
exec (Seq c1 c2) = exec c2 . exec c1
exec (While2 c) = fix f
  where
    f g s = if length s < 2
      then s
      else g (exec c s)

fix f = f (fix f)

Correct handling of while may be enough for an excellent solution.

In the actual exam, you do not have to write so much code necessarily. Some parts may be given with elisions indicated.
“Define a program analysis for a given problem and language.”
“Define a program analysis for a given problem and language.”

For a simple imperative syntax, as shown, please check whether programs *may* loop in the sense that they contain while loops with bodies lacking any assignments. The test for a statement to lack assignments is given already. Implement the function mayLoop.

```haskell
data Stm = Skip |
            Assign String Exp |
            Seq Stm Stm |
            If Exp Stm Stm |
            While Exp Stm

data Exp = ...

lacksAssign :: Stm -> Bool
mayLoop :: Stm -> Bool

lacksAssign Skip = True
lacksAssign (Assign _ _) = False
lacksAssign (Seq s1 s2) = lacksAssign s1 && lacksAssign s2
lacksAssign (If _ s1 s2) = lacksAssign s1 && lacksAssign s2
lacksAssign (While _ s) = lacksAssign s
```
Solution

mayLoop Skip = False
mayLoop (Assign _ _) = False
mayLoop (Seq s1 s2) = mayLoop s1 || mayLoop s2
mayLoop (If _ s1 s2) = mayLoop s1 || mayLoop s2
mayLoop (While _ s) = lacksAssign s || mayLoop s

In the actual exam, you should typically write less code than shown above. To this end, some trivial cases may be prepared for you.
Rather than defining a complete analysis, the assignment may also be concerned with an important building block of an analysis -- specifically a complete lattice.

For instance, define a partial order, \textit{leq}, for the complete lattice of special Booleans with least element Bottom, greatest element Top and incomparable values in between : True' and False'.

\begin{verbatim}
data Bool' = Bottom | True' | False' | Top
\end{verbatim}
Solution

\[
\begin{align*}
\text{leq} & :\ Bool' \to Bool' \to Bool \\
\text{leq Bottom } & = \text{True} \\
\text{leq } & \text{Top } = \text{True} \\
\text{leq True'} & = \text{True} \\
\text{leq False'} & = \text{True} \\
\text{leq } & = \text{False}
\end{align*}
\]
“Demonstrate declarative programming skills.”

Here you need to show good understanding of programming in Prolog and Haskell in the PLT context.
Define a predicate for membership test using `foldr`. Given a value `x` and a list `l` of values, the predicate determines whether `x` appears in `l`. 
Simple solution

```haskell
member :: Eq a => a -> [a] -> Bool
member x l = foldr f False l
  where
    f y r = x == y || r
```
Another solution

\[
\text{member :: Eq } \ 	ext{a} \Rightarrow \ a \rightarrow \ [\text{a}] \rightarrow \ \text{Bool} \\
\text{member } x = \text{or} \ . \ \text{map} \ (x==)
\]
Provide a function `skippy` that takes a list and returns a list with the even indexes of the list (starting to count at 0).

For example:

```python
skippy ["a","b","c","d"] = ["a","c"]
```

You can define the function any way you like.
Solution

skippy [] = []

skippy (x0:[]) = [x0]

skippy (x0:x1:xs) = x0:skippy xs
This type class models the extraction of all ints from a given value. **Add instances for lists and pairs.**

```haskell
class ToInts x where
toInts :: x -> [Int]

instance ToInts Int where
toInts i = [i]

instance ToInts Bool where
toInts = const []
```

“Demonstrate declarative programming skills.”
Solution

```haskell
instance ToInts a => ToInts [a]
  where
    toInts = concat . map toInts

instance (ToInts a, ToInts b) => ToInts (a,b)
  where
    toInts (a,b) = toInts a ++ toInts b
```
class Size x
  where
    size :: x -> Int

instance Size Bool
  where
    size = const 1

instance (Size a, Size b) => Size (a,b)
  where
    size (a,b) = size a + size b + 1

This type class counts constructors in terms. We assume that primitive values count as 1. Define instances for Int and lists.

“Demonstrate declarative programming skills.”
instance Size Int
    where
    size = const 1

instance Size a => Size [a]
    where
    size = (+) 1 . sum . map size

Instead (+1), we may also count length + 1 for the cons’es.

Less point-free code is also acceptable, but you may need to remember sum, map, and friends in order to quickly right down the solution.
Rephrase the following definition of append (which uses direct recursion) such that it uses Haskell’s fixed point combinator (also shown below):

\[
\text{append} :: [a] \rightarrow [a] \rightarrow [a] \\
\text{append} \ [\] \ l = \ l \\
\text{append} \ (h: t) \ l = h : \text{append} \ t \ l
\]

\[
\text{fix} :: (t \rightarrow t) \rightarrow t \\
\text{fix} \ f = f \ (\text{fix} \ f)
\]
Solution

\[
\text{append } l_1 \ l_2 = \text{fix} \ \text{append'} \ l_1 \\
\text{where} \\
\text{append'} \ [\ ] = l_2 \\
\text{append'} \ f \ (h:t) = h : f \ t
\]

This is already a complicated example because it involves a function with two parameters with some tricky order. You should count on something more straightforward.
Rephrase the following find function so that it takes a continuation to process the Int and to use an arbitrary result type.

```
find :: [(String, Int)] -> String -> Maybe Int
find [] s = Nothing
find ((k,v):l) s = if k==s then Just v else find l s
```
Solution

\[
\text{find'} \; :: \; [(\text{String, Int})] \rightarrow \text{String} \rightarrow (\text{Int} \rightarrow r) \rightarrow \text{Maybe r}
\]

\[
\text{find'} \; [] \; x \; f = \text{Nothing}
\]

\[
\text{find'} \; ((x',v):t) \; x \; f = \text{if } x'==x \text{ then Just } (f \; v) \text{ else find'} \; t \; x \; f
\]

\[
> \text{find'} \; [("x",42),("y",88)] \; "x" \; \text{id}
\]

\[
\text{Just 42}
\]

\[
> \text{find'} \; [("x",42),("y",88)] \; "x" \; (+1)
\]

\[
\text{Just 43}
\]

\[
> \text{find'} \; [("x",42),("y",88)] \; "z" \; (+1)
\]

\[
\text{Nothing}
\]

In the actual exam, you are likely to get a bit more help (by means of a more detailed explanation, the type of find' etc.)
Consider the following functional interpreter for a tiny functional language. It exposes one bug: “Return i” does not properly terminate the computation. Fix this problem by migrating to continuation style.

```haskell
data Exp = Id | Plus1 | Dot Exp Exp | Return Int

eval :: Exp -> Int -> Int

eval Id = id
eval Plus1 = (+1)
eval (Dot f g) = eval f . eval g
eval (Return i) = const i

> eval (Dot Plus1 (Dot Plus1 (Return 1))) 42 3
Wrong result! We want “1” instead.
```

“Demonstrate declarative programming skills.”
Solution

\[\text{eval'} :: \text{Exp} \rightarrow (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}\]

\[\text{eval'} \ \text{Id} \quad = \quad \text{id}\]

\[\text{eval'} \ \text{Plus1} \quad = \quad \text{flip} \ (\.) \ (\text{+1})\]

\[\text{eval'} \ (\text{Dot} \ f \ g) \quad = \quad \text{eval'} g \ . \ \text{eval'} f\]

\[\text{eval'} \ (\text{Return} \ i) \quad = \quad \text{const} \ $ \ \text{const} \ i\]

> eval' (Dot Plus1 (Dot Plus1 (Return 1))) id 42 1

In the actual exam, you are likely to get a bit more help (by means of a more detailed explanation, the type of eval’ etc.)
Consider the following interpreter for integers with operations for "0", successor, and predecessor:

```haskell
data Expr = Zero | Succ Expr | Pred Expr deriving Show

eval :: Expr -> Int
eval Zero = 0
eval (Succ t) = eval t + 1
eval (Pred t) = eval t - 1
```

Here is the beginning of the same function in monadic style:

```haskell
eval :: Monad m => Expr -> m Int
eval Zero = return 0
```

Add the missing equations for `Succ` and `Pred`.

**Reference solution**

```haskell
eval (Succ t) = eval t >>= \v -> return (v + 1)
```

```haskell
eval (Pred t) = eval t >>= \v -> return (v - 1)
```

As a matter of motivation, this may be a great preparation for disallowing negative numbers, when applying the predecessor function. An outstanding solution could illustrate this potential.

Complete the following monadic variation of the interpreter:

```haskell
eval :: Monad m => Expr -> m Int
eval Zero = return 0
```
Solution

eval (Succ t) = eval t >>= \v -> return (v + 1)
eval (Pred t) = eval t >>= \v -> return (v - 1)
Haskell was exercised above, but Prolog could be required.
“Solve a PLT riddle with a succinct argument.”

Here you need to show deeper understanding of semantics, types, Prolog, and Haskell.
"Solve a PLT riddle with a succinct argument."

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathsf{ass}_{\text{ns}})</td>
<td>(\langle x := a, s \rangle \rightarrow s[x \mapsto A[a]]s)</td>
</tr>
<tr>
<td>(\mathsf{skip}_{\text{ns}})</td>
<td>(\langle \text{skip}, s \rangle \rightarrow s)</td>
</tr>
<tr>
<td>(\mathsf{comp}_{\text{ns}})</td>
<td>(\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s'' \quad \langle S_1 ; S_2, s \rangle \rightarrow s'')</td>
</tr>
<tr>
<td>(\mathsf{if}^\text{tt}_{\text{ns}})</td>
<td>(\langle S_1, s \rangle \rightarrow s' \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s') if (B[b]s = \text{tt})</td>
</tr>
<tr>
<td>(\mathsf{if}^\text{ff}_{\text{ns}})</td>
<td>(\langle S_2, s \rangle \rightarrow s' \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s') if (B[b]s = \text{ff})</td>
</tr>
<tr>
<td>(\mathsf{while}^\text{tt}_{\text{ns}})</td>
<td>(\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s'' \quad \langle \text{while } b \text{ do } S, s \rangle \rightarrow s'') if (B[b]s = \text{tt})</td>
</tr>
<tr>
<td>(\mathsf{while}^\text{ff}_{\text{ns}})</td>
<td>(\langle \text{while } b \text{ do } S, s \rangle \rightarrow s) if (B[b]s = \text{ff})</td>
</tr>
</tbody>
</table>

Which, if any, of these Natural Semantics rules for \textit{While} violate the principle of compositionality? If so, in what sense?
Solution

Importantly, we face the judgement for statement semantics. A compositional semantics needs to compose the semantics of a compound statement from the semantics of constituent statements. This rule is violated by the rule for loops because the rule refers to the semantics of the loop itself (under a different initial state).
“Solve a PLT riddle with a succinct argument.”

Why does it make sense to proof properties for compositional semantics by means of structural induction?
Simply speaking, a compositional semantics decomposes terms and and recurses into those components. Hence, we can use structural induction (induction on the size of terms); terms considered by premises are smaller than terms considered by the conclusion.
“Solve a PLT riddle with a succinct argument.”

What language construct benefits from the generality of SOS compared to Natural Semantics?
Solution

Parallel composition is more versatile with SOS because the operands may proceed in an interleaving manner as opposed to commitment to an operand, as it necessary in a Natural Semantics.
“Solve a PLT riddle with a succinct argument.”

What would be a super-trivial language with a type system and an SOS semantics such that type safety is violated?
Solution

Expressions: \( e ::= v \mid z \)
Values: \( v ::= x \mid y \)
Types: \( t ::= a \mid b \)
SOS axioms: \( z \rightarrow x \)
Typing axioms: \( x : a, y : b, z : b \)
Culprit: \( z \) because \( z : b \) but \( z \rightarrow x \) and \( x : a \)

No longer than this!
“Solve a PLT riddle with a succinct argument.”

How do we see that the given semantics for the lambda calculus is call-by-value?

$t$ proxies for general terms.
$v$ proxies for values (i.e., terms in normal form).

How do we see that the given semantics for the lambda calculus is call-by-value?
This is evident from the fact that beta-reduction is only applied once the argument position of a function application is in the value form.
“Solve a PLT riddle with a succinct argument.”

\[ \lambda x : ? . x \ x \]

How does come that self application (shown above) is not typeable in the simply-typed lambda calculus?
There are simple types and function types. In order for self-application to be typeable, we must have that the type of the argument of self-application equals the function type of self-application. Argument or result type of a function type is strictly a part of the latter.
“Solve a PLT riddle with a succinct argument.”

$t ::= x \mid v \mid tt \mid t[T]$

$v ::= \lambda x : T.t \mid \forall X.t$

$T ::= X \mid T \rightarrow T \mid \forall X.T$

System F provides type abstraction in a manner similar to function abstraction in the basic lambda calculus. Syntactically (and in fact, fundamentally), how do these constructs differ?
Solution

Small lambdas are associated with types. Big lambdas are not associated with any such constraint. (Why is that? Strange!)
“Solve a PLT riddle with a succinct argument.”

\[ p = \{\ast \text{nat}, \{a = 1, b = \lambda x:\text{nat.} \text{pred } x\}\} \text{ as } \exists X, \{a:X, b:X \to X\} \]

Why should we argue that the existentially quantified type of \( p \) is likely to be of no use?
Solution

The hidden type cannot be observed in any manner. That is, while \( b \) can be applied to \( a \) (or to a result of a previous application of \( b \)), there is no information that we can ever extract from \( a \) or any said application of \( b \).
“Solve a PLT riddle with a succinct argument.”

Assume that $T$ is a well-formed class table. If $e : \tau$ then either

1. $v$ value, or

2. $e$ has the form $(c) \textbf{new} d(e_0)$ with $e_0$ value and $d \not\subseteq c$, or

3. there exists $e'$ such that $e \rightarrow e'$. 

This is the progress part of the type-safety theorem for Featherweight Java. What does it say?
Solution

1. Expression evaluation may have reached a normal form. 2. Expression evaluation may have gotten stuck with an expression that applies a case to a normal form where the target type is not a subtype of the normal form’s type. 3. Expression evaluation may still make progress with one step.

No longer than this!
“Solve a PLT riddle with a succinct argument.”

1. $(\text{out } x \ y; \ P) \mid (\text{in } x \ (z); \ Q) \rightarrow P \mid Q[y/z]$
2. If $P \rightarrow Q$ then $P \mid R \rightarrow Q \mid R$.

These are the (two most important) SOS rules for the Pl-calculus. What happens if we face a composition ("\mid") with one process sending on channel $foo$ and the other one receiving on channel $bar$?
Solution

There is no rule that proceeds from such a composition. The composition gets stuck. The first rule does not apply because the channels are not the same for send and receive. The second rule does not apply because there is no way to proceed with a term that has a heading send or receive.
Logistics

• Length: 60 minutes (+ initial explanation and other overhead)

• No phones, computers, electronics, books, notes, etc.

• You must bring your student ID.

• Registration via KLIPS to the extent feasible.

• See website for admission rules.

• Reference solution will be published right after exam.

• Results will be communicated by email.
All the best for the exam!
Make sure to talk to the Software Languages Team about research projects.