x = 1

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Language Theory

Preparation for the Midterm

Ralf Lämmel

Lectures covered

- Big-Step Operational Semantics
- Small-Step Operational Semantics
- Type Systems
- The Untyped Lambda Calculus
- The Simply Typed Lambda Calculus
- Lambda Calculi With Polymorphism
- Featherweight Java
- Concurrency Calculi

Underlying principles

- Heavily based on formal and/or executable notation.
  - ✦ “No text”, “No Multiple Choice”
  - ✦ Rule-based systems can be presented in Prolog.
  - ✦ Phantasy greek notation is acceptable as well.
- Based on subjects/skills covered by assignments.
- Many concepts and intuitions from lecture needed.

Categories of questions for ***midterm***
(0-2 questions per category; 6 questions in total)

1. Implement the **abstract syntax** of given constructs **in Prolog**.
2. Define a **compositional semantics** for given constructs.
3. Define a **natural semantics** for given constructs.
4. Define an **SOS semantics** for given constructs.
5. Define a **type system** for given constructs.
6. Give a **derivation tree** for a given term and given rules.
7. Solve a **semantics riddle** with a succinct argument.

!

Languages in scope:
- While
- B/NB
- λ cube
- CCS/π
- Java
- Prolog
...

What to expect from the ***final***?

- Denotational semantics in addition to operational semantics.
- Program analysis in addition to semantics.
- Haskell-centric instead of Prolog-centric.
- Advanced programming techniques in Haskell.
  - ✦ Monads
  - ✦ ...

1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-
13	-
14	-
15	4,0
16	3,7
17	3,7
18	3,3
19	3,3
20	3,0
21	2,7
22	2,7
23	2,3
24	2,3
25	2,0
26	1,7
27	1,7
28	1,3
29	1,3
30	1,0
31	1,0
32	1,0

Grading rules

- One final grade
- 0-2 points per question
  - ✦ 0 "missing or mental assault"
  - ✦ 1 "the beginning of an idea"
  - ✦ 2 "nearly or fully complete/correct"
- 1 possible extra point per exam
  - ✦ for an "outstanding solution"
- 6 questions for midterm (12 points, 40 %)
- 9 questions for final (18 points, 60 %)
- 30 points in total + 2 extra points

Samples questions and answers

Category

"Implement the abstract syntax of given constructs in Prolog."

“Implement 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*.**

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

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

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

Type application

Type abstraction

Polymorphic type

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).
```

One needs domain knowledge
regarding categories and constructs!

“Implement 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. Please add a short explanation per clause so that all combinators are named. You can leave out restriction, relabeling, and definitions of agent constants. You may also take a predicate action/1 for actions for granted.

Solution

```
term(seq(A,T)) :- action(A), term(T). % sequential combinator
term(T1+T2) :- term(T1), term(T2). % summation
tem(T1|T2) :- term(T1), term(T2). % composition
```

One needs domain knowledge regarding syntax (and elsewhere semantics)! A proposal is correct, even if names are slightly different.

“Implement 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.

Solution

```
sequence([]).
sequence([H|T]) :- op(H), sequence(T).
op(push(X)) :- number(X).
op(add).
```

One needs to observe informal elements (such as operation sequencing) 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 grouping while the intention is to limit the representation to sequences of ops. Nevertheless, this would still be considered a “good solution”.

Category

“‘Define a compositional semantics for given constructs.’”

“Define a compositional semantics for given constructs.”

```
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).
```

The result of expression evaluation may be a Boolean or a number value.

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 compositional semantics for given constructs.”

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.

Solution

You don't need to know / mention that part!

```
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).
```

Category

“Define a natural semantics for given constructs.”

“Define a natural semantics for given constructs.”

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 for given constructs.”

(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:

$$\begin{array}{rll} s(s(z)) & \textit{evaluates to} & 2 \\ x = s(s(z)) & \textit{evaluates to} & 2 \\ (x = s(s(z)), s(v(x))) & \textit{evaluates to} & 3 \end{array}$$

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

```
lookup(M,X,Y) :- append(_,[(X,Y)|_],M).

update([],X,Y,[(X,Y)]).
update([(X,_)|M],X,Y,[(X,Y)|M]).
update([(X1,Y1)|M1],X2,Y2,[(X1,Y1)|M2]) :-
 \+ X1 = X2,
 update(M1,X2,Y2,M2).
```

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
```

Category

“Define an SOS semantics for given constructs.”

“Define an SOS semantics for given constructs.”

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.
```

“Define an SOS semantics for given constructs.”

Consider a trivial programming language *Hyphen* which can essentially print any number of hyphens. This language has the following constructs: *skip* (i.e., the empty program), sequential composition (possibly denoted by “(...,...)”), *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 “*ntimes*(N,...)”). Here is an illustrative execution in Prolog:

```
?- manysteps(ntimes(7,hyphen),[],Output).
Output = [-, -, -, -, -, -, -].
```

Devise the step/4 relation for *Hyphen*.

Solution

```
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.
```

Category

“Define a type system for given constructs.”

"Define a type system for given constructs."

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.

```
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).
```

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

Possibly outstanding solution

```
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 for given constructs.”

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.

Solution

```
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

```
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).
```

Category

"Give a derivation tree for a given term and given rules."

“Give a derivation tree for a given term and given rules.”

Typing rules

true : Bool

false : Bool

0 : Nat

$$\frac{x : Nat}{s(x) : Nat}$$

$$\frac{x : Nat, y : Nat}{x + y : Nat}$$

Term

0 + s(s(0))

Solution

$$\frac{0 : Nat \qquad \dfrac{\dfrac{0 : Nat}{s(0) : Nat}}{s(s(0)) : Nat}}{0 + s(s(0) : Nat}$$

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)`**

“Give a derivation tree for a given term and given rules.”

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

```
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).
```

Give a typing derivation for the following term:

$$succ(if(iszero(zero),zero,succ(zero)))$$

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)

Category

“Solve a semantics riddle with a succinct argument.”

“Solve a semantics riddle with a succinct argument.”

$[\text{ass}_{\text{ns}}]$	$\langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\text{skip}_{\text{ns}}]$	$\langle \texttt{skip},\ s\rangle \to s$
$[\text{comp}_{\text{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$
$[\text{if}^{\text{tt}}_{\text{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{ns}}]$	$\dfrac{\langle S_2,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}^{\text{tt}}_{\text{ns}}]$	$\dfrac{\langle S,\ s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}^{\text{ff}}_{\text{ns}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Which, if any, of these Natural Semantics rules for *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).

No longer than this!

"Solve a semantics riddle with a succinct argument."

Why does it make sense to proof properties for compositional semantics by means of structural induction?

Solution

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.

No longer than this!

“Solve a semantics 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.

No longer than this!

“Solve a semantics 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 | z
Values	v := x | y
Types	t ::= a | b
SOS axioms	z -> x
Typing axioms	x : a, y : b, z : b
Culprit:	z because z : b but z -> x and x : a

No longer than this!

"Solve a semantics riddle with a succinct argument."

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$(\lambda x.t)\ v \longrightarrow [v/x]t$$

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

Solution

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.

No longer than this!

“Solve a semantics 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?

Solution

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.

No longer than this!

"Solve a semantics riddle with a succinct argument."

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

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

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

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!)

No longer than this!

"Solve a semantics riddle with a succinct argument."

$p = \{{*}nat, \{a = 1, b = \lambda x{:}nat.\ pred\ x\}\}$ ***as*** $\{\exists X, \{a{:}X, b{:}X \rightarrow X\}\}$

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*.

No longer than this!

“Solve a semantics 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* $(c)\,\mathtt{new}\,d(\underline{e_0})$ *with* e_0 value *and* $d \not\trianglelefteq c$, *or*
3. *there exists* e' *such that* $e \mapsto e'$.

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 semantics riddle with a succinct argument."

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

These are the (two most important) SOS rules for the PI-calculus. What happens if we face a composition ("|") 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.

No longer than this!

st

Logistics

- *10.00 am, 21 Dec 2010, Room E114.*
- No phones, computers, electronics, books, notes, etc.
- You must bring your student ID.
- No need to formally register / deregister.
- Everyone is admitted to the midterm.
- Admission rules to final see website.
- Your attendance only counts if you attend the final exam.
- 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 me about research projects.