INJE08: Programming Paradigms
04IN1024: Programming Language Theory
Resit SS 2014

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel, M.Sc. Andrei Varanovich
28.05.2014

Name, Vorname	
Matrikel-Nr.	
Studiengang	
ECTS (8 oder 6)	

Alle Fragen zählen für die 8 ECTS-Version. 2 Fragen zählen nicht für die 6 ECTS-Version; die Auswahl maximiert die Gesamtpunktzahl.

Korrekturabschnitt

Aufgabe	**Punkte (0-2)**	**Zusatzpunkt?**
1		
2		
3		
4		
5		
6		
7		
8		

1 "Define the semantics of substitution."

You surely remember the simple untyped lambda calculus with no extensions whatsoever. Here is its syntax again:

```
// Lambda terms
M ::= x | \x.M | M M
// Variables
x
```

Define substitution $[t/v]\,t'$ of a variable v by a term t within term t'. For instance, $[(\lambda x.\ x)/y]\,(z\,y) = z\,(\lambda x.\ x)$.

Reference solution

$$[N/x]x = N$$
$$[N/x]y = y, y \text{ any variable different from } x$$
$$[N/x](M_1\,M_2) = ([N/x]M_1)\,([N/x]M_2)$$
$$[N/x](\lambda x.M) = \lambda x.M$$
$$[N/x](\lambda y.M) = \lambda y.([N/x]M), y \text{ not free in } N$$

(The case in need of alpha conversion is not defined here.)

2 "Define the syntax of CCS."

Cover at least the structure of labels including 'tau', prefixing, and composition.

Reference solution

```
% Unbared and bared names
label(nobar(X)) :- atom(X).
label(bar(X)) :- atom(X).

% Actions
action(X) :- label(X).
action(tau). % Completed perfect action

% Syntax of processes
expr(prefix(A, E)) :- action(A), expr(E). % prefixing
expr(comp(E1, E2)) :- expr(E1), expr(E2). % concurrent composition
...
```

3 "Calculate a derivation tree."

Consider the following big-step semantics of some expression forms:

```
eval(_, const(I), I).
eval(S, var(X), V) :- member((X, V), S).
eval(S, binary(E1, E2, add), V) :- eval(S, E1, V1), eval(S, E2, V2), V is V1 + V2.
eval(S, binary(E1, E2, mul), V) :- eval(S, E1, V1), eval(S, E2, V2), V is V1 * V2.
```

Write down the derivation tree for the following expression:

```
binary(binary(var(x), const(22), add), const(1), mul)
```

You need to pick a suitable state so that expression evaluation succeeds.

Reference solution

```
* eval( [(x,20)],
        binary(binary(var(x), const(22), add), const(1), mul),
        42)
  * eval([(x,20)], binary(var(x), const(22), add), 42)
    * eval([(x,20)], var(x), 20)
      * member((x,20), [(x,20)])
    * eval([(x,20)], const(22), 22)
    * 42 is 20 + 22
  * eval([(x,20)], const(1), 1)
  * 42 is 42 * 1
```

4 "Convert big-step into denotational style."

Convert the semantics of the previous task into denotational style. *Notation: Define the semantics in an appropriate denotational semantics notation. You may use Haskell, too.*

Reference solution

```
-- The syntax in Haskell for clarity (not required)
data Expr = Const Int | Var String | Binary Expr Expr Op
data Op = Add | Mul

-- Structure of states for clarity (not required)
type State = String -> Int

-- Meaning of expressions
eval :: Expr -> State -> Int
eval (Const i) _ = i
eval (Var x) s = s x
eval (Binary e1 e2 o) s = op o (eval e1 s) (eval e2 s)

-- Meaning of operator symbols
op :: Op -> Int -> Int -> Int
op Add = (+)
op Mul = (*)
```

5 "Define well-formedness for an imperative language."

Consider the following syntax of expression and statement forms:

```
% Expression forms
expr(const(N)) :- number(N). % Constant expression
expr(var(X)) :- atom(X). % Variable reference
expr(read). % Read a number from standard input

% Statement forms
stm(skip). % Empty statement
stm(assign(X, E)) :- atom(X), expr(E). % Variable assignment
stm(seq(S1, S2)) :- stm(S1), stm(S2). % Statement sequence
stm(if(E, S1, S2)) :- expr(E), stm(S1), stm(S2). % If-then-else
```

Define a well-formedness judgement that checks that variables are not overridden. For instance, the following statement is not well-formed because x is first assigned '1' and then '2':

```
seq(assign(x, const(1)), assign(x, const(2)))
```

Notation: Define the judgement(s) in a notation appropriate for operational semantics, well-formedness, or type systems. You may use Prolog or Haskell, too. (You can take for granted standard predicates on lists or sets such as member/2, union/3, intersection/3.)

Reference solution

```
ok(skip, []).
ok(assign(X, _), [X]).
ok(seq(S1, S2), L3) :-
  ok(S1, L1),
  ok(S2, L2),
  intersection(L1, L2, []), % intersection must be empty
  union(L1, L2, L3).
ok(if(_, S1, S2), L3) :-
  ok(S1, L1),
  ok(S2, L2),
  % intersection(L1, L2, []), % intersection may be non-empty
  union(L1, L2, L3).
```

An approach with a threaded argument for assigned variables and a simple membership test per assignment is also Ok.

6 "Define the small-step semantics of while loops."

Just assume a straightforward syntax for while loops—as in $while(E, S)$, where E is an expression and S is a statement. Expression evaluation observes a state; statement execution transforms a state. *Notation: Define the semantics in an appropriate operational semantics notation. You may use Prolog or Haskell, too.*

Reference solution

```
exec(while(E, S), M, skip, M) :−
  eval(E, M, false).

exec(while(E, S), M, seq(S, while(E, S)), M) :−
  eval(E, M, true).
```

7 "Explain the simplifcations assumed by Featherweight Java."

Consider again the syntax of Featherweight Java:

Classes	C	::=	$\texttt{class}\, c\, \texttt{extends}\, c'\, \{\underline{c\, f}\texttt{;}\, k\, \underline{d}\}$
Constructors	k	::=	$c\texttt{(}\underline{c\, x}\texttt{)}\, \{\texttt{super(}\underline{x}\texttt{);}\, \underline{\texttt{this.}f\texttt{=}x}\texttt{;}\}$
Methods	d	::=	$c\, m\texttt{(}\underline{c\, x}\texttt{)}\, \{\texttt{return}\, e\texttt{;}\}$
Types	τ	::=	c
Expressions	e	::=	$x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\underline{e}\texttt{)}$
			$\mid \texttt{new}\, c\texttt{(}\underline{e}\texttt{)} \mid \texttt{(}c\texttt{)}\, e$

Describe (in 42 words or less) the deliberate restrictions imposed on Java by Featherweight Java.

Reference solution

Fields are initialized during construction, but never assigned. There are no interfaces. There are no primitive types. There are no generics. There are no static methods. ...

8 "Show that you understand type safety."

Consider the following trivial syntax of expression forms for Booleans *true* and *false* and natural numbers with *zero*, *successor*, and *addition*.

```
% Syntax of terms
term(true).
term(false).
term(zero).
term(succ(T)) :- term(T).
term(add(T1, T2)) :- term(T1), term(T2).

% Values (normal forms)
value(true).
value(false).
value(zero).
value(succ(T)) :- value(T).
```

Describe an operational semantics for evaluation together with a type system so that Booleans and natural numbers are not confused, i.e., type safety is guaranteed. *Notation: Define the judgement(s) in a notation appropriate for operational semantics, well-formedness, or type systems. You may use Prolog or Haskell, too.*

Reference solution

```
% Types of terms
type(true, bool).
type(false, bool).
type(zero, nat).
type(succ(T), nat) :- type(T, nat).
type(add(T1, T2), nat) :- type(T1, nat), type(T2, nat).

% Operational (big-step) semantics
eval(true, true).
eval(false, false).
eval(zero, zero).
eval(succ(T), V2) :- eval(T, V1), V2 is V1 + 1.
eval(add(T1, T2), V3) :- eval(T1, V1), eval(T2, V2), V3 is V1 + V2.
```