INJE08: Programming Paradigms
04IN1024: Programming Language Theory
Final WS 2013/14

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel, M.Sc. Andrei Varanovich
05.02.2014

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 the lambda calculus."

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 the semantics of lambda terms. (You can take substitution for granted.) *Notation: Define the semantics in an appropriate operational semantics notation. You may use Prolog or Haskell, too.*

Reference solution

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

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

2 "Define the semantics of CCS composition."

You surely remember CCS. Here is some part of CCS syntax and semantics:

```
% 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
...

% Judgements:
% - negate(A1, A2): Negation of labels
% - step(E1, A, E2): Initial process, action, resulting process
```

You only need to define the semantics of the `comp`(..., ...) form. *Notation: Define the semantics in an appropriate operational semantics notation. You may use Prolog or Haskell, too.*

Reference solution

```
% Left step for composition
step(comp(E1,E2), A, comp(E3,E2)) :-
  step(E1, A, E3).

% Right step for composition
step(comp(E1,E2), A, comp(E1,E3)) :-
  step(E2, A, E3).

% Composition with communication
step(comp(E1, E2), tau, comp(E3, E4)) :-
  step(E1, A1, E3),
  step(E2, A2, E4),
  negate(A1, A2).
```

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(const(20), const(22), add), const(1), mul)
```

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

Reference solution

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

4 "Convert big-step into small-step style."

Convert the semantics of the previous task into small-step style. Make sure to explicitly define normal forms. Also, make sure that the semantics is deterministic (such that each expression has at most one associated derivation sequence). *Notation: Define the semantics in an appropriate operational semantics notation. You may use Prolog or Haskell, too.*

Reference solution

```
% Values (normal forms)
value(const(_)).

eval(State, var(X), V) :-
  member((X, V), State).

eval(State, binary(E1, E2, O), binary(E3, E2, O)) :-
  eval(State, E1, E3).

eval(State, binary(E1, E2, O), binary(E1, E3, O)) :-
  value(E1),
  eval(State, E2, E3).

eval(State, binary(E1, E2, add), const(V)) :-
  E1 = const(V1),
  E2 = const(V2),
  V is V1 + V2.

eval(State, binary(E1, E2, mul), const(V)) :-
  E1 = const(V1),
  E2 = const(V2),
  V is V1 * V2.
```

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 all variable references are definitely preceded by an assignment in the control flow. For instance, the following statement is not well-formed because y is referenced before it is assigned a value:

```
seq(assign(x, var(y)), assign(y, read))
```

Notation: Define the judgement in a notation appropriate for 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(const(N), L, L).
ok(var(X), L, L) :- member(X, L).
ok(read).
ok(skip).
ok(assign(X, E), L1, L3) :- ok(E, L1, L2), union(L2, [X], L3).
ok(seq(S1, S2), L1, L3) :- ok(S1, L1, L2), ok(S2, L2, L3).
ok(if(E, S1, S2), L1, L5) :-
  ok(E, L1, L2),
  ok(S1, L2, L3),
  ok(S2, L2, L4),
  intersection(L3, L4, L5).
```

6 "Define the denotational semantics of while loops."

Just assume a straightforward syntax for while loops—as in **while** E **do** 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 denotational semantics notation. You may use Haskell, too.*

Reference solution

```
exec (While e s)
  = fix f
    where
      f g = cond (eval e) (g . exec s) id

-- Helper (not required)
cond a b c s =
  case (a s) of
    (BoolValue True) -> b s
    (BoolValue False) -> c s

-- Helper (not required)
fix f = f (fix f)
```

7 "Explain the normal forms for 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) a suitable definition of values (normal forms) for Featherweight Java.

Reference solution

Intuitively, objects are values in this language. Thus, the expression form for object construction provides the normal form—assuming that the constructor arguments are in normal form, too.

8 "Show that you understand type safety."

Consider the following trivial syntax of two constant terms x and y and the following typing judgement with two types a and b.

```
% Syntax of terms
term(x).
term(y).

% Values (normal forms)
value(x).

% Syntax of types
type(a).
type(b).

% Types of terms
typeOf(x,a).
typeOf(y,b).
```

Discuss (in 42 words or less) whether a semantics with type safety is possible.

Reference solution

Progress requires that y must be reduced. If y is reduced to x, this violates preservation because x and y are of different types. y can be reduced to y, though, thereby satisfying all of type safety vacuously.