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.*

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.*

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.

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.*

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.)

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.*

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.

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.