*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, $[(\lambda x.\ x)/y]\,(z\,y) = z\,(\lambda x.\ x)$.

## 2 "Define the syntax of CCS."

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

## 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.*

## 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.)

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

Just assume a straightforward syntax for while loops—as in $\mathit{while}(E, S)$, 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.*

## 7 "Explain the simplifcations assumed by Featherweight Java."

Consider again the syntax of Featherweight Java:

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

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

## 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.*