INJE08: Programming Paradigms
04IN1024: Programming Language Theory

Final WS 2015/16

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel
10.02.2016

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

Grading scheme

There are 9 questions with 0-2 points each: 2 points for a (mostly) correct and complete solution; 1 point for a somewhat reasonable solution with significant omissions or defects; 0 points otherwise. Points/grade: 0-8/5, 9/4, 10/3.7, 11/3.3, 12/3, 13/2.7, 14/2.3, 15/2, 16/1.7, 17/1.3, 18/1. Old exam rules: 0-10:5, 11:4, 12:3.7, 13:3.3, 14:3, 15:2.7, 16:2, 17:1.7, 18:1.

Contents

1 Topic: *Syntax*

Define the abstract syntax of *function definitions with function signatures* in a simple functional language like BFPL. You can ignore the details of the syntax of types and expressions. Use ESL or Haskell or Prolog for the abstract syntax definition.

2 Topic: *Interpreter*

Consider the data type *Command* for operations on a stack according to the type alias *Stack*, as interpreted by the function *interpret*:

```
*Main> :i Command
data Command = Empty | Push Int Command | Pop Command
*Main> :i Stack
type Stack = [Int]
*Main> :t interpret
interpret :: Command -> Stack
*Main> interpret (Pop (Push 88 (Push 42 Empty)))
[42]
```

Define the *interpret* function.

3 Topic: *Big-step operational semantics*

Consider the following axiom for expression evaluation:

$$\mathsf{true} \;\rightarrow\; \mathsf{true} \qquad [\mathsf{TRUE}]$$

Argue why this axiom would be part of a big-step as opposed to a small-step semantics.

4 Topic: *Small-step operational semantics*

Consider the following rule of a big-step operational semantics for the function application in the basic functional programming language BFPL.

$$\frac{\begin{array}{c}\phi, \eta \vdash e_1 \rightarrow v_1 \quad \cdots \quad \phi, \eta \vdash e_k \rightarrow v_k \\ \langle n, sig, \langle \langle n_1, \ldots, n_k \rangle, e \rangle \rangle \in \phi \\ \phi, [n_1 \mapsto v_1, \ldots, n_k \mapsto v_k] \vdash e \rightarrow v\end{array}}{\phi, \eta \vdash \mathsf{apply}(n, [e_1, \ldots, e_k]) \rightarrow v} \quad [\mathsf{APPLY}]$$

Explain, in one concise sentence, how the rule(s) of the small-step semantics would differ from the shown rule of the big-step semantics.

5 Topic: *Lambda calculus*

Remember substitution $[N/x]M$, with M the term in which to replace all free occurrences of variable x by the term N. What is the result of $[z/x](\lambda z.\ x\ z)$?

6 Topic: *Type systems*

Consider the following type system and small-step operational semantics. There are terms t, values v and types τ. A term can be of the forms a, b, and $\mathsf{c}(t)$. There are two values: a and b. There are two types: q and r.

Type system

$$\mathsf{a} : \mathsf{q} \quad [\mathsf{type1}]$$

$$\mathsf{b} : \mathsf{r} \quad [\mathsf{type2}]$$

$$\frac{t : \mathsf{r}}{\mathsf{c}(t) : \mathsf{q}} \quad [\mathsf{type3}]$$

Semantics

$$\frac{t \rightarrow t'}{\mathsf{c}(t) \rightarrow \mathsf{c}(t')} \quad [\mathsf{sem1}]$$

$$\mathsf{c}(\mathsf{a}) \rightarrow \mathsf{a} \quad [\mathsf{sem2}]$$

Explain, in one concise sentence, how the key properties of progress or preservation or both are violated.

7 Topic: *Polymorphism*

Define the syntax of System F. (Pretty much any grammar notation is acceptable: ESL or EGL, the use of Haskell or Prolog, or any grammar-like notation used on this course's slides or recommended textbooks.)

8 Topic: *Denotational semantics*

Consider the following Haskell-based denotational style semantics of statements in the basic imperative programming language BIPL.

```
execute :: Stmt -> Store -> Store
execute Skip = id
execute (Assign x e) = assign x (evaluate e)
execute (SequComp s1 s2) = execute s2 . execute s1
execute (IfThenElse e s1 s2) = ifThenElse (evaluate e) (execute s1) (execute s2)
execute (While e s) = while (evaluate e) (execute s)
```

Explain or specify (in Haskell or semi-formally) the combinator *ifThenElse*.

9 Topic: *Partial evaluation*

In a program specializer or an inliner, we need to be able to turn expressions into values and vice versa:

```
-- These functions are part of the BFPL inliner.
exprToValue :: Expr -> Maybe Value
valueToExpr :: Value -> Expr
```

Explain, in a concise sentence, why the first function is partial (as indicated by the use of *Maybe*), whereas the second function is total.