

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: <i>Syntax</i>	3
2	Topic: <i>Interpreter</i>	4
3	Topic: <i>Big-step operational semantics</i>	5
4	Topic: <i>Small-step operational semantics</i>	6
5	Topic: <i>Lambda calculus</i>	7
6	Topic: <i>Type systems</i>	8
7	Topic: <i>Polymorphism</i>	9
8	Topic: <i>Denotational semantics</i>	10
9	Topic: <i>Partial evaluation</i>	11

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.

Reference solution

```
type function = (atom, funsig, fundef) ;  
type funsig = (type*, type) ;  
type fundef = (atom*, expr) ;
```

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.

Reference solution

```
data Command = Empty | Push Int Command | Pop Command
type Stack = [Int]

interpret :: Command -> Stack
interpret Empty = []
interpret (Push i s) = i : interpret s
interpret (Pop s) = tail (interpret s)
```

3 Topic: *Big-step operational semantics*

Consider the following axiom for expression evaluation:

$\text{true} \rightarrow \text{true}$ [TRUE]

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

Reference solution

The modeled evaluation does not involve a small (or any) step, as the result equals the argument; only a big-step semantics models such ‘zero-step’ computations.

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.

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

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.

Reference solution

There would be a separate rule to make progress with arguments of the function application and the shown rule would not evaluate the arguments, but instead assume that they are in value form; neither would the rule evaluate the body of the definition, but instead return the result of the substitution as the result of the step.

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 $[z/x](\lambda z. x z)$?

Reference solution

$\lambda a. z a$

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 $c(t)$. There are two values: a and b . There are two types: q and r .

Type system

$a : q$ [type1]

$b : r$ [type2]

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

Semantics

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

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

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

Reference solution

Progress is violated because the form $c(t)$ will get stuck because [sem2] would require t to evaluate to a , but the type system insists on terms of type r , which rules out a .

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.)

Reference solution

$$t ::= x \mid v \mid t t \mid t[T]$$

$$v ::= \lambda x : T. t \mid \Lambda X. t$$

$$T ::= X \mid T \rightarrow T \mid \forall X. T$$

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*.

Reference solution

In Code (less details are sufficient):

```
-- Results of expression evaluation
type Value = Either Int Bool

-- Stores as maps from variable ids to values
type Store = Map String Value

-- Store transformers
type StoreT = Store -> Store

-- Conditional computation
ifThenElse :: (Store -> Value) -> StoreT -> StoreT -> StoreT
ifThenElse f g h sto =
  let (Right v) = f sto in
    if v then g sto else h sto
```

In words: the first argument is the meaning of the condition, which, in the presence of a store, may or may not evaluate to a Boolean value, while the result is only defined for the former case based on dispatch on the Boolean value to either the meaning of the then or the else branch—both being applied to the store.

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.

Reference solution

We can assume that all value forms of our language—such as integer or boolean values in BFPL—are matched by expression forms—such as corresponding constant forms of expressions, but we cannot generally assume that all expression forms immediately correspond to a value, e.g., a binary expression needs to be evaluated before it could be possibly a value.