INJE08: Programming Paradigms
04IN1024: Programming Language Theory

Dry run WS 2015/16

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel
03.02.2016

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* **3**

2 Topic: *Interpreter* **4**

3 Topic: *Big-step operational semantics* **5**

4 Topic: *Small-step operational semantics* **6**

5 Topic: *Lambda calculus* **7**

6 Topic: *Type systems* **8**

7 Topic: *Polymorphism* **9**

8 Topic: *Denotational semantics* **10**

9 Topic: *Partial evaluation* **11**

The topics of the dry run are supposed to be representative of those to be expected from final and resit. Deviations would be announced. Of course, the topics are chosen to cover broad subjects areas. The actual questions are focused on specific competences that were emphasized in the lecture and exercised in the lab—in the edition of the course at hand.

1 Topic: *Syntax*

Define the concrete and abstract syntax of a language construct such that both syntaxes differ in a non-trivial way (i.e., other than by simply omitting terminals in abstract syntax). Add a comment about the assumed language and the reason for the difference.

Reference solution

This is about the if statement in an imperative language. The else branch is optional in the concrete syntax, whereas it is not optional in the abstract syntax because the empty statement can be assumed for a missing else branch.

Concrete syntax (in EGL: notation):

```
[if] stmt : 'if' '(' expr ')' stmt { 'else' stmt }? ;
```

Abstract syntax (in ESL notation):

```
symbol if : expr x stmt x stmt -> stmt ;
```

2 Topic: *Interpreter*

Consider the data type *Set* and the demonstration of the *elementOf* function which is assumed to interpret terms of type *Set* in the sense of set membership:

```
> :i Set
data Set = Const [Int] | Union Set Set | Difference Set Set
> :t elementOf
elementOf :: Set -> Int -> Bool
> elementOf (Difference (Union (Const []) (Const [1..100])) (Const [42])) 88
True
```

Define the *elementOf* function.

Reference solution

```
data Set = Const [Int] | Union Set Set | Difference Set Set

elementOf :: Set -> Int -> Bool
elementOf (Const is) i = elem i is
elementOf (Union s1 s2) i = elementOf s1 i || elementOf s2 i
elementOf (Difference s1 s2) i = elementOf s1 i && not (elementOf s2 i)
```

3 Topic: *Big-step operational semantics*

Consider the following rules of the big-step operational semantics for a fragment of the BTL language—it is obviously concerned with operations on natural numbers.

$$\mathsf{zero} \;\to\; \mathsf{zero} \qquad [\mathsf{ZERO}]$$

$$\frac{t \;\to\; n}{\mathsf{succ}(t) \;\to\; \mathsf{succ}(n)} \qquad [\mathsf{SUCC}]$$

$$\frac{t \;\to\; \mathsf{zero}}{\mathsf{pred}(t) \;\to\; \mathsf{zero}} \qquad [\mathsf{PRED1}]$$

$$\frac{t \;\to\; \mathsf{succ}(n)}{\mathsf{pred}(t) \;\to\; n} \qquad [\mathsf{PRED2}]$$

Draw a derivation tree that uses each rule exactly once.

Reference solution

$$\dfrac{\dfrac{\dfrac{\mathsf{zero} \to \mathsf{zero}}{\mathsf{succ}(\mathsf{zero}) \to \mathsf{succ}(\mathsf{zero})}\,[SUCC]}{\mathsf{pred}(\mathsf{succ}(\mathsf{zero})) \to \mathsf{zero}}\,[PRED2]}{\mathsf{pred}(\mathsf{pred}(\mathsf{succ}(\mathsf{zero}))) \to \mathsf{zero}}\,[PRED1]$$

4 Topic: *Small-step operational semantics*

Consider the following rules of the small-step operational semantics for the basic functional programming language BFPL.

$$\frac{\phi \;\vdash\; e_{i+1} \;\Rightarrow\; {e_{i+1}}'}{\phi \;\vdash\; \mathsf{apply}(n, [v_1, \ldots, v_i, e_{i+1}, \ldots, e_k]) \;\Rightarrow\; \mathsf{apply}(n, [v_1, \ldots, v_i, {e_{i+1}}', \ldots, e_k])} \quad [\mathsf{apply1}]$$

$$\frac{\langle n, sig, \langle \langle n_1, \ldots, n_k \rangle, e \rangle \rangle \in \phi}{\phi \;\vdash\; \mathsf{apply}(n, [v_1, \ldots, v_k]) \;\Rightarrow\; e[n_1 \mapsto v_1, \ldots, n_k \mapsto v_k]} \quad [\mathsf{apply2}]$$

Explain, in one concise sentence, how the semantics would need to change so that we switch from call-by-value (i.e., applying the function body to actual values) to call-by-name (i.e., passing unevaluated actual arguments).

Reference solution

The first rule would not be needed at all while the second rule would need to substitute argument names by expressions.

5 Topic: *Lambda calculus*

Describe substitution $[N/x]M$, with M the term in which to replace all free occurrences of variable x by the term N, in a concise manner—formally or informally. (You would need to make a case discrimination over the forms of lambda terms.)

Reference solution

$[N/x]x = N$

$[N/x]y = y, y$ any variable different from x

$[N/x](M_1\ M_2) = ([N/x]M_1)\,([N/x]M_2)$

$[N/x](\lambda x.M) = \lambda x.M$

$[N/x](\lambda y.M) = \lambda y.([N/x]M), y$ not free in N

If this condition is not met, then alpha conversion is needed.

6 Topic: *Type systems*

Consider the following type system and big-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		Semantics	
$\mathsf{a} : \mathsf{q}$	[type1]	$\mathsf{a} \to \mathsf{a}$	[sem1]
$\mathsf{b} : \mathsf{r}$	[type2]	$\mathsf{b} \to \mathsf{b}$	[sem2]
$\dfrac{t : \mathsf{r}}{\mathsf{c}(t) : \mathsf{q}}$	[type3]	$\dfrac{t \to v}{\mathsf{c}(\mathsf{t}) \to v}$	[sem3]

Explain, in one concise sentence, how the semantics fails to enforce a constraint put up by the type system.

Reference solution

The type system requires type r for subterm t and promises type q for the compound $\mathsf{c}(t)$ whereas the semantics may just accept a value of any type for t and return it as the result of $\mathsf{c}(t)$. To put it differently, type safety is violated.

7 Topic: *Polymorphism*

Please be reminded of the Haskell type of '(.)':

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Here is an application of '(.)':

```
Prelude> (.) not not True
True
```

What is the type of '(.)' in System F? How is the application represented in System F? (Hint: You need to add type application.)

Reference solution

Type of '(.)':

$$\forall a.\ \forall b.\ \forall c.\ (b \to c) \to (a \to b) \to a \to c$$

Term for application:

$$(.)[Bool][Bool][Bool]\ not\ not\ True$$

8 Topic: *Denotational semantics*

In an imperative language with a loop construct, compositionality dictates that the semantics of loops involves a fixed point construction. Explain, in a concise sentence, when (how, why) would fixed point construction be needed in the denotational semantics of a simple functional programming language (such as BFPL).

Reference solution

Recursive functions, as they are expressible in BFPL, would trigger the use of a fixed point construction because the application of any function would need to be mapped to the meaning of the corresponding function definition parametrized in the meaning of recursive applications.

9 Topic: *Partial evaluation*

Consider this rule for binary expressions, as it is part of an inliner for the basic functional programming language BFPL:

```
pevaluate (Binary o e1 e2) env =
  let
    r1 = pevaluate e1 env
    r2 = pevaluate e2 env
  in
    case (exprToValue r1, exprToValue r2) of
      (Just v1, Just v2) -> valueToExpr (binOp o v1 v2)
      _ -> Binary o r1 r2
```

Explain, in a concise sentence, how binary expressions are inlined.

Reference solution

If both operands can be partially evaluated to values, then the corresponding binary operation is applied, and a value is returned, and the partially evaluated operands are otherwise recombined in a binary expression.