INJE08: Programming Paradigms
04IN1024: Programming Language Theory

Dry run SS 2016

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel
13.07.2016

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

The topics of the dry run are supposed to be representative of those to be expected from final. Of course, the topics are chosen to cover broad subjects areas. The actual questions are focused on specific competences covered in the course and the underlying material.

1 Topic: *Syntax*

General remark regarding this question type: *You are typically asked to define some piece of concrete or abstract syntax preferably in the EGL and ESL notations of the course.*

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 # stmt # stmt -> stmt ;
```

2 Topic: *Interpretation*

General remark regarding this question type: *You are typically asked to implement some interpreter functionality in Haskell. You can also use Prolog, if you really wanted to. Interpreters are programs—so you must program here.*

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
```

Implement the *elementOf* function in Haskell.

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*

General remark regarding this question type: *You are typically asked to explain or to sketch some fragment of a big-step operational semantics for some simple language constructs or to draw a derivation tree for a simple program fragment. You should use the formal inference rule-based notation of the course.*

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*

General remark regarding this question type: *See the general remark for big-step style, which applies here accordingly to small-step style. Overall, these two questions types may also be used to make you demonstrate your understanding of the differences between big- and small-step style. Further, you may be asked to explain some characteristcs of the styles.*

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: *Denotational semantics*

General remark regarding this question type: *See the general remarks on big- and small-step style, which apply here accordingly—except that you would not get to see or use inference rule-based notation here, but rather the designated functional and equation-based notation for denotational semantics. Alternatively, Haskell may be used to represent 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.

6 Topic: *Abstract interpretation*

General remark regarding this question type: *You are likely to be asked to explain or define some detail of an abstract interpretation. It is unlikely that you would be asked to define a complete abstract interpretation, as this would be typically too complicated. We will focus on abstract interpretation for imperative languages, as discussed in the course.*

Consider the following

```
data AbstractBool = Bottom | True' | False' | Top
```

Define the least-upper bound operation on `AbstractBool`.

Reference solution

```
data AbstractBool = Bottom | True' | False' | Top

lub :: AbstractBool -> AbstractBool -> AbstractBool
lub Bottom x = x
lub x Bottom = x
lub False' False' = False'
lub True' True' = True'
lub _ _ = Top
```

7 Topic: *Lambda calculus*

General remark regarding this question type: *You are likely to be asked about the syntax, the semantics and type system of the lambda calculus. This also includes the notion of substitution. Generally, you need to understand the meaning of simple lambda terms (such as '$\lambda\, x.\ x$' to correspond to the identity function).*

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.

8 Topic: *Type systems*

General remark regarding this question type: *You are typically asked to explain or to sketch some fragment of a type system for some simple language constructs or to draw a typing derivation for a simple program fragment. You should use the formal inference rule-based notation of the course.*

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]	$\mathsf{a} \rightarrow \mathsf{a}$	[sem1]
$\mathsf{b} : \mathsf{r}$	[type2]	$\mathsf{b} \rightarrow \mathsf{b}$	[sem2]
$\dfrac{t : \mathsf{r}}{\mathsf{c}(t) : \mathsf{q}}$	[type3]	$\dfrac{t \rightarrow v}{\mathsf{c}(\mathsf{t}) \rightarrow v}$	[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.

9 Topic: *Polymorphism*

General remark regarding this question type: *You are typically asked about universal (parametric) polymorphism and possibly structural polymorphism (subtyping). More specifically, you may get to see or use system F and the record types, as discussed in the course. Just like with all the other question types, a specific question does not necessarily use or require use of formal notation—instead, general 'informal' questions regarding your understanding of the involved notions may be there, too.*

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:

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