INFOMCPD Concepts of program design 2022–2023
Sample Exam
2023

Name:

Student number:

Please read the following instructions carefully:

- Fill in your name and student number above. Be prepared to identify yourself with your student card when you submit your exam.
- You are allowed to bring two single sided or one double sided A4 sheet of handwritten notes. You are forbidden from accessing any other external material, additional notes, electronic material, or online resources.
- Answer each open question in the space provided, if possible. Otherwise, use the space on the last pages marked 'Overflow'. Write clearly and legibly.
- A maximum of 100 points can be obtained by the questions on this exam. The final score is calculated by dividing the total number of points scored by 10.

Please do not write in the space below.

Question	Points	Score
Part 1: Multiple Choice and Short Answer Questions	30	
Part 2: Inference rules and induction	25	
Part 3: Semantics, Procedural Programming	20	
Part 4: Abstract Machines	25	
Total:	100	

Part 1: Multiple Choice and Short Answer Questions

(a) (3 points) Which of the following statements are **incorrect** (multiple answers possible)?

- ◯ If a set S is inductively defined by a set R of inference rules, then $s \in S$ implies $s\ S$ is derivable using the rules in R.
- ◯ If a set S is inductively defined by a set R of inference rules, then $s\ S$ is derivable with the rules only if s is an element of the set S.
- ◯ If a set S is inductively defined by a set R of inference rules, then there is only one way to derive $s\ S$ using the rules in R.
- ◯ If a set S is inductively defined by a set R of inference rules, then removing any rule from R would mean that there exists at least on element s in S, for which we cannot derive $s\ S$ anymore using only the remaining rules.
- ◯ If a set S is inductively defined by a set R of inference rules, then R contains exactly one axiom.
- ◯ None of the above.

(b) (3 points) Which of the following statements are **incorrect** (multiple answers possible)?

- ◯ A parser translates the concrete syntax representation of a program into an internal representation in abstract syntax.
- ◯ A parser translates the first-order abstract syntax representation of a program into an internal representation in higher-order abstract syntax.
- ◯ Higher-order abstract syntax is necessary for programming languages which support higher-order functions.
- ◯ Higher-order abstract syntax has a built-in notion of variables.
- ◯ None of the above.

(c) (3 points) Which of the following statements are **incorrect** (multiple answers possible)?

- ◯ Most programming languages, including C# and Haskell, support dynamic scoping.
- ◯ Only object oriented language support dynamic scoping.
- ◯ In a type-safe language, all computations have to terminate.
- ◯ To show that a language is type safe, it is sufficient to prove progress and preservation.
- ◯ Dynamically typed languages can be type safe.

(d) (3 points) Which of the following statements are **incorrect** (multiple answers possible)?

- ◯ Small step semantics rules are in general more detailled than big step sematics rules.
- ◯ It is not possible to model evaluation order with big step semantics.
- ◯ Both small step and big step semantics are so-called operational semantics.
- ◯ Function closures are necessary when we want to implement a language with partial application of functions using environment semantics.
- ◯ Function closures are necessary when we want to implement a language with partial application of functions with explicit control flow.
- ◯ None of the above.

(e) (3 points) Give an example of two non-identical, but α-equivalent MinHs expressions.

(f) (6 points) Give the most general type of the following polymorphic, implicitly typed MinHs expressions, if it exists. If not, explain briefly why.

1. `InL (InR 5)`

2.
```
recfun f x = case x of
               InL y -> y
               InR z -> z
```

3.
```
recfun cmp g =
  recfun cmp' f =
    recfun cmp'' x = g (f x)
```
or, equivalently, if we can have multiple parameters to functions:

```
recfun cmp g f x = g (f x)
```

(g) (6 points) For which of the following types can we write a total, terminating, polymorphic MinHs function? If such a function exists, provide the definition, otherwise just write "No".

1. `(a * b) -> (a + b)`

2. `() -> (a + b)`

3. `(a -> b) -> (b -> a)`

(h) (3 points) Consider the following Haskell type constructor:

```
data Mysterious a  = Mysterious (a -> a)
```

Is `Mysterious` co-, contra, or invariant with respect to subtyping?

Part 2: Inference rules and induction

The set strings of strings B is inductively defined by the following set of rules:

$$\frac{}{\texttt{Z}\ B}(B1) \qquad \frac{s\ B}{s\texttt{I}\ B}(B2) \qquad \frac{s\texttt{I}\ B}{s\texttt{0}\ B}(B3)$$

(a) (3 points) Using the inference rules given above, show that $\texttt{Z0I}\ B$ is derivable.

(b) (5 points) Is the following rule derivable, admissible, or not admissible with respect to the inductive definition of B given above? Explain briefly.

$$\frac{s\ B}{s\texttt{0}\ B} \tag{1}$$

(c) (3 points) The function *value* maps strings in B to natural numbers:

$$\begin{aligned} value\ (\texttt{Z}) &= 0 \\ value\ (s\texttt{0}) &= 2 * value(s) \\ value\ (s\texttt{I}) &= 2 * value(s) + 1 \end{aligned}$$

Examples:

$$\begin{aligned} value\ (\texttt{ZI}) &= 1 \\ value\ (\texttt{ZI0}) &= 2 \\ value\ (\texttt{ZII}) &= 3 \\ value\ (\texttt{ZI0I}) &= 5 \end{aligned}$$

The following inference rules define a relation $\oplus \in B \times B$, such that $(x \oplus y)$ is derivable if and only if $value(y) = value(x) + 1$.

$$\frac{}{\texttt{Z} \oplus \texttt{ZI}} \qquad \frac{s\ B}{s\texttt{0} \oplus s\texttt{I}} \qquad \frac{s \oplus s'}{s\texttt{I} \oplus s'\texttt{0}}$$

Show that $\texttt{ZII} \oplus \texttt{ZI00}$ is derivable.

(d) (8 points) Using rule induction, show that for all strings s and s', $s \oplus s'$ implies

$$value(s) + 1 = value(s')$$

(e) (6 points) Provide the inference rules to define a relation $\lhd \in B \times B$, such that $(x \lhd y)$ is derivable if and only if $value(y) < value(x)$. You can define it using $\oplus$, or from scratch.

Part 3: Semantics, Procedural Programming

Consider the following language, TinyW, which is a simple subset of TinyC. A program here is just a statement, which can contain, of course, many other statements:

prgm ::= *stmt*
vdecs ::= ϵ | *vdec vdecs*
vdec ::= `int` *Ident* `=` *v* `;`
stmt ::= *expr* `;` | `{` *vdecs stmts* `}` | `while (` *expr* `)` *stmt*
stmts ::= ϵ | *stmt stmts*
expr ::= *v* | *Ident* | $expr_1$ `+` $expr_2$ | *Ident* `=` *expr*

The statics semantics checks that all variables in the program are in scope, and it works like the static semantics rules for TinyC.

The big step semantics is also a simpler version of TinyC's semantics. Statements and expression are evaluated in the context of a store (or environment) state to a result integer value and environment. The initial state consists of an initially empty environment and the main statement.

Looking up a variable is written $g@x = 5$, where x is a variable in the environment g, and 5 is the result of the lookup. To set the value of the variable x to 5 we write $g@x \leftarrow 5$, and $g.int\ x = 5$ to add a new declaration $int\ x = 5$ to the environment g. Lookup retrieves the rightmost occurence, and update also affects only the rightmost occurence of a variable. For example, $(int\ x = 5.int\ x = 10)@x$ results in the value 10, and $(int\ x = 5.int\ x = 10)x \leftarrow 20$ in the environment $(int\ x = 5.int\ x = 20)$.

$$\frac{(g,e) \Downarrow (g', v_1) \qquad (g', e_2) \Downarrow (g'', v_2)}{(g, e_1{+}e_2) \Downarrow (g'', v_1 + v_2)}\ addition \qquad \frac{(g.l, ss) \Downarrow (g'.l', v)}{(g, \{l\ ss\}) \Downarrow (g', v)}\ blocks$$

$$\frac{(g,s) \Downarrow (g', v) \quad (g', ss) \Downarrow (g'', v')}{(g, s\ ss) \Downarrow (g'', v')}\ sequence\ of\ statements \qquad \frac{}{(g, \epsilon) \Downarrow (g, 0;)}\ empty\ sequence$$

$$\frac{g@x = v}{(g, x) \Downarrow (g, v)}\ variables \qquad \frac{(g,e) \Downarrow (g'.v)}{(g, x{=}e) \Downarrow (g'@x \leftarrow v, v)}\ assignment$$

$$\frac{(g,e) \Downarrow (g', 0)}{(g, \texttt{while(}e\texttt{)}s) \Downarrow (g', 0;)}\ while\text{-}1 \qquad \frac{(g,e) \Downarrow (g', v), v \neq 0 \qquad (g', s;\ \texttt{while(}e\texttt{)}s) \Downarrow (g'', v)}{(g, \texttt{while(}e\texttt{)}s) \Downarrow (g'', v)}\ while\text{-}2$$

(a) (8 points) We add another type of expression to the language, namely *postfix increment*:

$$expr ::= Ident\texttt{++}$$

It increments the value of variable *Ident* by one, and evaluates to the **original** value of the variable (i.e., value before the increment).

Provide the big step dynamic semantics rule for postfix increment.

(b) (6 points) Are the following two code snippets semantically equivalent (that is, result in the same state and value) for every statement s?

```
while (x++)
   s
```

and

```
while (x)
   {x = x + 1; s}
```

Explain your answer briefly.

(c) (6 points) When we added reference types to TinyC, we introduced a second type of storage in addition to the stack g, namely the heap h.

Why was this necessary? Give an example program which would not evaluate properly if we store both the reference and the referenced `int` value in g.

Part 4: Abstract Machines

Consider the boolean expression language Ex, whose higher-order abstract syntax is given below:

$$\frac{c \in \{\texttt{True}, \texttt{False}\}}{c\ Ex} \qquad \frac{e_1\ Ex \quad e_2\ Ex}{(\texttt{Or}\ e_1\ e_2)\ Ex} \qquad \frac{e\ Ex}{(\texttt{Not}\ e)\ Ex} \qquad \frac{e\ Ex}{(\texttt{Exists}\ (x.e))\ Ex} \qquad \frac{}{x\ Ex}$$

whose big step semantics is provided below:

$$\frac{}{\texttt{True} \Downarrow \texttt{True}} \qquad \frac{}{\texttt{False} \Downarrow \texttt{False}} \qquad \frac{e_1 \Downarrow \texttt{True}}{(\texttt{Or}\ e_1\ e_2) \Downarrow \texttt{True}} \qquad \frac{e_1 \Downarrow \texttt{False} \quad e_2 \Downarrow v}{(\texttt{Or}\ e_1\ e_2) \Downarrow v}$$

$$\frac{e \Downarrow \texttt{False}}{(\texttt{Not}\ e) \Downarrow \texttt{True}} \qquad \frac{e \Downarrow \texttt{True}}{(\texttt{Not}\ e) \Downarrow \texttt{False}}$$

$$\frac{(\texttt{Or}\ (e[x := \texttt{True}])\ (e[x := \texttt{False}])) \Downarrow v}{(\texttt{Exists}\ (x.e)) \Downarrow v}$$

(a) (6 points) Evaluate the expression

```
(Exists  (x. (Not x)))
```

using the semantics above. Provide the derivation.

(b) (7 points) Is the language type safe? Explain your answer briefly.
Note: you can view Ex as (only) type here. That is, every expression e for which $e\ Ex$ is derivable has the type Ex.

(c) (12 points) Provide an equivalent small step environment semantics in the style of the E-machine. That is, all the rules should be axioms, and it should handle variables via environments, not substitution.

State clearly what are the possible initial states of the machine, and the possible final states are.

Overflow