INFOMCPD Concepts of program design 2019–2020
Retake Exam
April 16th, 9:00 – 12:00

Name:

Student number:

Please read the following instructions carefully:

- When you are finished, but no later than 12:00, send the document with your answers to `g.k.keller@uu.nl`.
- You can ask questions by contacting me via MS Teams. If necessary, I'll announce clarifications via the team chat for the retake exam.
- This is an open book exam. You can also use online resources, but you have to work on your own: do not communicate with anyone about the exam during the exam, and do not post questions on message boards and the like (apart from the MS Teams chat for this exam).
- 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.

Question	Points	Score
Static and Dynamic Semantics Basics, Abstract Machines	34	
Evaluation	24	
Abstract Machines	42	
Total:	100	

Static and Dynamic Semantics Basics, Abstract Machines

Marloes wants to design a simple language, which is essentially λ-calculus with integers and addition. She defines a judgement $\Gamma \vdash e$ `ok`, to denote that an expression e is a valid expression of the language under environment Γ in the abstract syntax:

$$\frac{n \in Int}{\Gamma \vdash (\textbf{Num}\ n)\ \texttt{ok}} \qquad \frac{x \in \Gamma}{\Gamma \vdash x\ \texttt{ok}} \qquad \frac{\Gamma \cup x \vdash e\ \texttt{ok}}{\Gamma \vdash (\textbf{Lambda}\ x.e)\ \texttt{ok}}$$

$$\frac{\Gamma \vdash e_1\ \texttt{ok} \quad \Gamma \vdash e_2\ \texttt{ok}}{\Gamma \vdash (\textbf{Apply}\ e_1\ e_2)\ \texttt{ok}} \qquad \frac{\Gamma \vdash e_1\ \texttt{ok} \quad \Gamma \vdash e_2\ \texttt{ok}}{\Gamma \vdash (\textbf{Plus}\ e_1\ e_2)\ \texttt{ok}}$$

She writes down the following rules for the transition system of small step semantics of the language:

$$\frac{}{(\textbf{Apply}\ (\textbf{Lambda}\ x.e_1)\ e_2) \mapsto e_1[x := e_2]} \qquad \frac{e_1 \mapsto e_1'}{(\textbf{Apply}\ e_1\ e_2) \mapsto (\textbf{Apply}\ e_1'\ e_2)}$$

$$\frac{}{(\textbf{Plus}\ (\textbf{Num}\ n_1)\ (\textbf{Num}\ n_2)) \mapsto (\textbf{Num}\ n_1 + n_2)} \qquad \frac{e_1 \mapsto e_1'}{(\textbf{Plus}\ e_1\ e_2) \mapsto (\textbf{Plus}\ e_1'\ e_2)}$$

$$\frac{e \mapsto e'}{(\textbf{Plus}\ (\textbf{Num}\ n)\ e) \mapsto (\textbf{Plus}\ (\textbf{Num}\ n)\ e')}$$

All expressions e for which $\{\} \vdash e$ `ok` is derivable are possible initial states (I), and expressions of the form (**Num** n) and (**Lambda** $x.e$), such that $\{\} \vdash$ (**Lambda** $x.e$) `ok` is derivable are possible final states (F).

(a) (4 points) What is the set S of all states (that is, initial, final *and* intermediate) of the abstract machine?

(b) (4 points) Give the full evaluation sequence of the form $e_1 \mapsto e_2 \mapsto \ldots \mapsto v$ where $v \in F$ for the expression

$$(\textbf{Apply}\ (\textbf{Lambda}\ x.(\textbf{Plus}\ x\ (\textbf{Plus}\ x\ x)))\ (\textbf{Num}\ 4))$$

(c) (4 points) Marloes realises that the static semantics is too weak: that is, there are expressions for which $\{\} \vdash e$ `ok` is derivable, but for which the evaluation will end up in a stuck state.

Give two example expressions for which this happens. The two example expressions should get stuck for different reasons (that is, caused by different rules or the lack thereof).

> **Solution:** There is no type checking, so expressions which result in functions as argument to addition, or non-function values at the first argument of an apply will get stuck.

(d) (8 points) Add rules to the dynamic semantics to fix the problem. Expressions which do not get stuck according to the original semantics should still evaluate to the same result. Adjust the definition of S, F and I if necessary.

> **Solution:** Introduce an error value as legal value. Add evaluation rules which map problematic additions or function applications to error. The error value propagates through the evaluation (not necessary to write out every single rule of the error propagation).

(e) (14 points) Define a new static semantics such that the language is type safe, leave the dynamic semantics as Marloes defined it. Your rules should not be unnecessarily restrictive and allow as many evaluable expressions as possible while still being decidable.

Solution: Static semantics needs to take type of values into account. Note that it is not sufficient to have a single type for all function types. The type needs to encode both the argument and result type of each function, which can in turn be function types (see typing rules for MinHs. Simpler in this case, as we have no recursive functions).

Evaluation

Marloes asks Kees to implement an evaluator for the language using the single step semantics rules of the previous question as specification. To be more precise, given an expression e which is valid according to the static semantics, the evaluator should return `Just` v if $e \mapsto^! v$. In addition, the implementation should use an environment, not substitution, for efficiency reasons.

Given an expression which is not valid, it should return `Nothing`. Invalid expressions are expressions for which $\{\} \vdash e$ `ok` can't be derived in the original static semantics or which lead to a stuck state.

Kees shows her his implementation (see code below). Marloes tests Kees' implementation and realises that the behaviour of the program does not match the specification for some of her test cases.

(a) (8 points) Find the bugs in the code and give expressions (in Haskell syntax or abstract syntax of the spec) for which the behaviour of the program differs from the specification

Solution:

- Addition does not return `Nothing` is applied to type incorrect arguments, but causes a run-time error
- Any invalid expression in a function which is then not applied does not evaluate to error, or functions with a free variable which get applied in a context where this variable gets caught (this is easy to miss).
- Closures are missing (with the usual effect that partical application of functions can lead to the escape of bound variables.

(b) (16 points) Fix the bugs in the code. You do not need to worry about minor syntactic errors. You can solve this question on paper or on your computer.
Hint: Fixing one of the bugs requires substantial changes to the code.

```
data LExpr
  = Num    Int
  | Var    String
  | Plus   LExpr   LExpr
  | Lambda String  LExpr
  | Apply  LExpr   LExpr
  deriving (Show)


type Env = [(String, LExpr)]

eval :: Env -> LExpr -> Maybe LExpr
eval env (Num n) = Just (Num n)
eval env (Var x) = lookup x env
eval env (Lambda x e) = Just (Lambda x e)
eval env (Apply e1 e2)
  = case (eval env e1, eval env e2) of
      (Just (Lambda x e), Just val) -> eval ((x, val) : env) e
      (_                , _)        -> Nothing
eval env (Plus e1 e2)
  = case (eval env e1, eval env e2) of
      (Just (Num n1), Just (Num n2)) -> Just (Num (n1 + n2))
```

Solution:

- Add extra case to return error value for incorrect application of addition
- Add scope checking
- Add closures

Abstract Machines

Consider the TinyC language we defined in the lecture. The version below is simplified (no conditionals, no returns, simpler expressions, functions have only one argument), but it also includes support for arrays of integers.

prgm ::= *gdecs stmt*
gdecs ::= ϵ | *gdec gdecs*
gdec ::= *fdec* | *vdec*
vdecs ::= ϵ | *vdec vdecs*
vdec ::= `int` *Ident* `=` v `;` | `int[]` *Ident* `=` `newArray` `(`v_1, v_2`);`
fdec ::= *type Ident* `(`*type* $Ident_2$`)` *stmt*
type ::= `int` | `int[]`
stmt ::= *expr* `;` | `{` *vdecs stmts* `}` | `while` `(` *expr* `)` *stmt*
stmts ::= ϵ | *stmt stmts*
expr ::= *Num* | *Ident* | *Ident* `[` *expr*`]` | $expr_1$ `+` $expr_2$ | *Ident* `=` *expr* | *Ident* `[` *expr*`]` `=` *expr* | *Ident* `(`*expr*`)`

Variable declarations (*vdec*) can now also be of the form

```
int [ ] a = newArray (10, 5);
```

which declares a new array `arr` of size 10, initialised with the value `5` at each position.

Expressions (*expr*) can now also be array access

```
a[0]
```

indicating the access to the first element of array `a`, and array update

```
a[0]  = 10
```

setting the first value of array `a` to 10.

(a) (12 points) Provide the static semantics rules for this language. You can base them on the ones given in the lecture (TinyC lecture, pdf also on MS Teams) and adapt it to handle arrays.
It will be it necessary to change the definition of the variable environment (V) and function environment (F).
The static semantics should ensure that operations are only applied to expressions and variables of the correct type.

Solution: We need to distinguish between int and array types. Functions don't need to store arity, but argument and result type:

- Type τ are either *int* or *int []*
- Environment of variables: $V = \{x_1 : \tau_1, x_2 : \tau_2, ...\}$
- Environment of functions with their argument and result type: $F = f1 : (\tau_1, tau'_1, f2 :, ...$

New rules (others stay the same):

$$\frac{V \vdash \tau\ x = e\ \textbf{decs}\ \{x : \tau\} \qquad V \cup \{x : \tau\}, F \vdash gdecs\ \textbf{decs}\ V', F'}{V, F \vdash \tau\ x = e; gdecs\ \textbf{decs}\ V' \cup \{x : \tau\}, F'}$$

$$\frac{f \notin F \qquad V, F \cup \{f : (\tau_1, \tau_2), x : \tau_1\} \vdash stmt : \tau_2 \qquad V, F \cup \{f : (\tau, Int)\} \vdash gdecs \textbf{ decs } V', F'}{V, F \vdash \tau_2\ f(\tau_1\ x) = stmt; gdecs \textbf{ decs } V', F' \cup \{f : (\tau_1, \tau_2)\}}$$

All the expression and statement rules are typed instead of just **ok**: indexing allowed only on arrays, while and if require int as expression type of conditional, and for the latter, the same type for both branches. Here are some of the rules with **ok** replaced by the concrete type:

$$\frac{x : \tau \in V}{F, V \vdash x : \tau}$$

$$\frac{F, V \vdash expr : \tau \quad x : \tau \in V}{F, V \vdash x = expr : \tau}$$

$$\frac{F, V \vdash expr : \tau_1 \quad f : (\tau_1, \tau_2) \in F}{F, V \vdash f(expr) : \tau_2}$$

(b) (10 points) Extend the dynamic semantics rules to handle arrays. For this subquestion, arrays should be passed by value to a function. That is, a copy of **the whole array** is passed to a function. Similarly, if an array variable is assigned a new (array typed), as in this example

```
int [ ] a = newArray (10, 5);
int [ ] b = newArray (20, 10);

a = b;
```

the array on the right hand side of the assignment should be copied.

Make sure the resulting language is still type safe.

Solution: extending environment with an array value: we need to store the size of the array as well as the elements. An easy way to do this is to add the declaration to the environment in the same way as for values: we write $g.int\ []\ x = (4, 10)$ adds the information that `x` is of size 4. We then add a rule which puts $int\ x[0] = 10$ to `int x[4] = 10` on the environment stack.

When indexing an array, either on the left hand side of an assignment, or in an expression, we need to add a range check to the rule.

Updating a whole array updates the size information. We can just add entries $a[i] = b[i]$ to the environment stack. It does not matter that this means that the old values of a are still around, since they are shadowed by the new ones or out of range.

(c) (10 points) In C (and most other non-functional languages), passing arrays to a function is done by reference, similarly for assignments.

That is, in the example below, the value of `b[0]` would be `100` after the function call is executed.

```
int f (int [ ] a)
  a[0] = 100;


int [ ] b = newArray (20, 10);
f (b);
```

If we changed our TinyC language to pass by reference for arrays (and for array assignment), in which way would this complicate the formalisation of the language? Which of the rules would 'break'?

Solution: Pass by value works just like assigment, where we copy the whole array. This doesn't work anymore, as changes to array values in the function would not alter the values of the array passed into the function.

(d) (10 points) Formalise the dynamic semantics for pass by reference. It is sufficient to describe what the new machine state would look like, and to give the rules for array declaration, array update, function calls and array indexing.

Solution: One solution is to associate each array name not just with its size, but also a fresh name (used no where else in the program). Say, for an array declaration $int\ [\ ]\ a\ =\ newArray\ (4,\ 10)$ we store the size 4, as well as a new name aX with the array, and put the values $int\ aX[i]\ =\ 10$ on the environment stack (instead of $a[0]\ =\ 10$). The new name acts as a reference. When this array is passed by reference into a function, the formal parameter needs to be associate in the environment with the size and the reference name.

Array declaration rule change: use new name to store array entries instead of array name. Store the new name together with the array size.

Array update and array indexing: look up the new name associated and size with the array, and operate on that.

Function call: associate the name of the formal parameter with the size and new name of the actual parameter. No need to copy anything else.