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).

(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.

(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.

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

(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))
```

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.

(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.

(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'?

(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.