CSED321 Assignment 8 - *Abstract Machine K*

Due Tuesday, May 21

In this assignment, you will implement the operational semantics of a programming language called Tiny ML (*TML*). TML is essentially a subset of OCaml; that is, every TML program is also a valid OCaml program. TML is powerful enough to support most of the functional features of OCaml. TML supports polymorphic functions, recursive functions, recursive datatypes, and pattern matching. Although TML does not directly support the `if-then-else` clause and other constructs, it is not difficult to emulate them with those constructs available in TML.

1 Fun with TML

The following TML program "list.tml" reverses an integer list, which demonstrates recursive datatypes, recursive functions, and pattern matching.

```
type list = Nil | Cons of (int * list) ;;

let rec append = fun l ->
  match l with
    Nil -> (fun x -> Cons (x, Nil))
  | Cons (h, t) -> (fun x -> Cons (h, append t x)) in

let rec reverse = fun l ->
  match l with
    Nil -> Nil
  | Cons (h, t) -> append (reverse t) h in

let l = Cons (1, Cons (2, Nil)) in
reverse l
```

The following TML program "array.tml" implements a functional array, which demonstrates mutable references and pattern matching.

```
let create = fun _ -> ref (fun i -> 0) in
let access = fun a -> fun i -> (! a) i in
let update = fun a -> fun i -> fun n ->
  let old = !a in
    a := fun j -> match i = j with true -> n | false -> old j in

let arr = create () in
let _ = update arr 1 3 in
access arr 1
```

The following TML program "poly.tml" takes two functions and returns their composition, which demonstrates polymorphic functions.

```
fun f -> fun g -> fun x -> g (f x)
```

2 Syntax

The concrete grammar of TML is defined as follows:

$$
\begin{array}{rl}
\textit{sconty} ::= & \textbf{int} \mid \textbf{bool} \mid \textbf{unit} \\
\textit{sconpat} ::= & \textit{num} \mid \textbf{true} \mid \textbf{false} \mid () \\
\textit{scon} ::= & \textit{num} \mid \textbf{true} \mid \textbf{false} \mid () \\
\textit{op} ::= & + \mid - \mid * \mid = \mid < \\
\textit{ty} ::= & \textit{sconty} \mid \textit{tycon} \mid \textit{ty} * \textit{ty} \mid \textit{ty} \texttt{ -> } \textit{ty} \mid \textit{ty}\ \textbf{ref} \mid (\textit{ty}) \\
\textit{pat} ::= & \_ \mid \textit{sconpat} \mid \textit{vid} \mid \textit{cid}\ \langle \textit{pat} \rangle \mid (\textit{pat}, \textit{pat}) \mid (\textit{pat}) \mid (\textit{pat} : \textit{ty}) \\
\textit{mrule} ::= & \textit{pat} \texttt{ -> } \textit{exp} \\
\textit{mlist} ::= & \textit{mrule}\ \langle \texttt{|}\ \textit{mlist} \rangle \\
\textit{exp} ::= & \textit{scon} \mid \textit{vid} \mid \textit{cid} \mid \textit{exp}\ \textit{exp} \mid \textit{exp}\ \textit{op}\ \textit{exp} \mid (\textit{exp}, \textit{exp}) \\
\mid & \textbf{ref}\ \textit{exp} \mid !\, \textit{exp} \mid \textit{exp} := \textit{exp} \mid \textbf{fun}\ \textit{mrule} \mid \textbf{match}\ \textit{exp}\ \textbf{with}\ \textit{mlist} \\
\mid & \textbf{let}\ \textit{pat} = \textit{exp}\ \textbf{in}\ \textit{exp} \mid \textbf{let rec}\ \textit{pat} = \textit{exp}\ \textbf{in}\ \textit{exp} \mid (\textit{exp}) \mid (\textit{exp} : \textit{ty}) \\
\textit{conbinding} ::= & \textit{cid}\ \langle \textbf{of}\ \textit{ty} \rangle \\
\textit{conbind} ::= & \textit{conbinding}\ \langle \texttt{|}\ \textit{conbind} \rangle \\
\textit{dec} ::= & \textbf{type}\ \textit{tycon} = \textit{conbind} \\
\textit{dlist} ::= & \langle \textit{dec} \rangle * \\
\textit{program} ::= & \langle \textit{dlist}\ ;; \rangle\ \textit{exp}
\end{array}
$$

All grammar variables (nonterminals) are written in *italic* (e.g., *pat* and *exp* are grammar variables). The meaning of each nonterminal is defined in a corresponding grammar rule. The only exceptions are *num*, *vid*, *cid*, and *tycon*: (i) *num* denotes an integer constant, a non-empty sequence of decimal digits; and (ii) *vid*, *cid*, and *tycon* denote an identifier, a sequence of letters, digits, apostrophes ' and underscore _, where *vid* and *tycon* begin with a lowercase letter, and *cid* begins with an uppercase letter. All TML reserved words are written in boldface (e.g., **let**, **ref** and **fun** are reserved words). Each grammar variable is read as follows:

- *num*: number
- *vid*: variable identifier
- *cid*: value constructor
- *tycon*: type constructor
- *sconty*: special constant type
- *sconpat*: special constant pattern
- *scon*: special constant
- *op*: primitive operation on integers
- *ty*: type
- *pat*: pattern
- *mrule*: match rule
- *mlist*: match rule list
- *exp*: expression
- *conbinding*: constructor binding
- *conbind*: constructor binding list
- *dec*: declaration
- *dlist*: declaration list
- *program*: TML program

Each grammar rule consists of a grammar variable on the left side and its expansion forms on the right side. Expansion forms are separated by $|$. For example, *scon* can be expanded into *num*, **true**, **false** or (). A pair of brackets $\langle\rangle$ encloses an optional phrase. For example, *conbinding* is expanded into either *cid* or *cid* followed by **of** *ty*. Starred brackets $\langle\rangle*$ represent zero or more repetitions of the enclosed phrase. For example, *dlist* can have zero or more *dec*'s. Notice that a declaration list and an expression in a top-level program is separated by `;;` (for parsing in OCaml). There are further syntactic restrictions:

- No *pat* can have the same *vid* twice. E.g., (x, y) is a legal pattern, whereas (x, x) is not.
- '**let rec** *pat* = *exp*' can only create functions. That is, *exp* must have the form '**fun** *mrule*'.
- No '**let** *pat* = *exp*' can bind **true** or **false** as patterns.
- No *conbinding* can have duplicate *cid*'s. E.g., **type** $t = A \mid A$ **of int** is not valid.

The file `tml.ml` defines types for the grammar variables in the structure `Tml`. The file `validate.ml` defines functions for syntactic validation. Specifically, the function `vprogram` takes a `Tml.program` value, checks whether the `Tml.program` value obeys the syntactic restrictions, and returns the `Tml.program` value. It raises `AstValidateError` if the `Tml.program` value violates the syntactic restrictions. Finally, the file `print.ml` defines functions for printing the content of grammar variables in the structure `Print`. The meaning of each function in the structure should be clear; for example, `exp2str exp` returns a string representation of the `Tml.exp` value `exp`. These functions can be useful when debugging your code.

3 Semantics

The goal of this assignment is to define the operational semantics of TML in the $\mathbb{K}$ framework. More precisely, we want to design and implement the abstract machine K with eager evaluation (i.e., call-by-value) strategies for TML. Because OCaml uses eager evaluation and any TML program is an OCaml program, you can use OCaml to execute TML programs, and your abstract machine K should give the same result as OCaml!

We show a partial definition of the abstract machine K below. A configuration consists of a computation k, an environment η, and a store ψ. A computation is a list of computation labels ϕ, which include expressions and values, evaluation contexts, and special purpose labels such as $\mathit{restore}(\eta)$. An environment η is a map from variables x to locations l, and a store ψ is a map from locations l to values v, where locations and closures are also values.

$$
\begin{array}{rrcl}
\textsf{value} & v & ::= & \mathit{scon} \mid \ldots \mathit{to\ be\ filled\ by\ students} \ldots \mid l \mid \mathit{closure}(\mathit{mrule}, \eta) \\
\textsf{environment} & \eta & ::= & \cdot \mid x \hookrightarrow l,\, \eta \\
\textsf{store} & \psi & ::= & \cdot \mid l \hookrightarrow v,\, \psi \\
\textsf{label} & \phi & ::= & \square\, e \mid v\, \square \mid \ldots \mathit{to\ be\ filled\ by\ students} \ldots \mid \mathit{restore}(\eta) \\
\textsf{computation} & k & ::= & e \mid k \curvearrowright \phi \\
\textsf{configuration} & \chi & ::= & \langle k \rangle_c\ \langle \eta \rangle_e\ \langle \psi \rangle_s
\end{array}
$$

The definition of environments has changed, compared to one presented in the class. The range of environment η is no longer values v, but *locations* l. Therefore, in order to find a value of variable x, you need both environment η and store ϕ. Also, the closure of a function involves only environments, *not stores.* The corresponding rules are as follows.

$$
\langle\ \frac{x \curvearrowright k}{v \curvearrowright k}\ \rangle_c\ \langle\ x \hookrightarrow l,\ \eta\ \rangle_e\ \langle\ l \hookrightarrow v,\ \psi\ \rangle_s
\qquad\qquad
\langle\ \frac{\textbf{fun}\ \mathit{mrule} \curvearrowright k}{\mathit{closure}(\mathit{mrule}, \eta) \curvearrowright k}\ \rangle_c\ \langle\ \eta\ \rangle_e
$$

For pattern matching, we consider a *pattern matching judgement* of the form $\sigma \vdash v \preceq p$, meaning that pattern p matches value v by a set of substitutions σ. The inference rules for the pattern matching judgement are defined as follows,

$$\frac{}{\sigma \vdash v \preceq \_}\mathit{Any} \qquad \frac{}{\sigma \vdash n \preceq n}\mathit{Num} \qquad \frac{}{\sigma \vdash b \preceq b}\mathit{Bool} \qquad \frac{}{\sigma \vdash () \preceq ()}\mathit{Unit}$$

$$\frac{v/x \in \sigma}{\sigma \vdash v \preceq x}\mathit{Var} \qquad \frac{}{\sigma \vdash c \preceq c}\mathit{Con} \qquad \frac{\sigma \vdash v \preceq p}{\sigma \vdash c\ v \preceq c\ p}\mathit{Capp}$$

$$\frac{\sigma \vdash v_1 \preceq p_1 \quad \sigma \vdash v_2 \preceq p_2}{\sigma \vdash (v_1, v_2) \preceq (p_1, p_2)}\mathit{Pair} \qquad \frac{\sigma \vdash v \preceq p}{\sigma \vdash v \preceq (p : \mathit{ty})}\mathit{Pty}$$

where n denotes a number, b denotes a Boolean value, x denotes a variable identifier, c denotes a value constructor, and ty denotes a type.

Remark. The above inference rules do *not* consider type declarations **type** *tycon* = *conbind*. A type system of TML—which will be implemented in the next assignment—will deal with whether patterns and values have valid types according to the type declarations.

Suppose that the computation label $\mathit{matcher}(v, p \to e)$ reduces to (σ, e) if $\sigma \vdash v \preceq p$, and $\perp$ othwerwise. The transition rules for pattern matching are straightforward as follows:

$$\langle\ \frac{\textbf{match}\ v\ \textbf{with}\ p \to e \mid \mathit{ml}\ \curvearrowright\ k}{\mathit{matcher}(v, p \to e)\ \curvearrowright\ \textbf{match}\ v\ \textbf{with}\ \mathit{ml}\ \curvearrowright\ k}\ \rangle_c$$

$$\langle\ \frac{\perp\ \curvearrowright\ \textbf{match}\ v\ \textbf{with}\ \mathit{ml}\ \curvearrowright\ k}{\textbf{match}\ v\ \textbf{with}\ \mathit{ml}\ \curvearrowright\ k}\ \rangle_c$$

$$\langle\ \frac{(\sigma, e)\ \curvearrowright\ \textbf{match}\ v\ \textbf{with}\ \mathit{ml}\ \curvearrowright\ k}{\mathit{bind}(\sigma)\ \curvearrowright\ e\ \curvearrowright\ \mathit{restore}(\eta)\ \curvearrowright\ k}\ \rangle_c\ \langle\ \eta\ \rangle_e$$

The label $\mathit{bind}(\sigma)$ allocates new locations for the values in σ and binds the variables in σ to the corresponding locations. The rules for function applications can be defined in a similar way.

We will not provide complete transition rules in this assignments. You will have to design and implement your own K transition rules, including rules for $\mathit{matcher}(v, p \to e)$, $\mathit{bind}(\sigma)$, function applications, **let** and **let rec** expressions, arithmetic and relational operators, etc.

Remark. You can use the same rules presented in the class for mutable references. However, the rules for function applications and recursive functions cannot be the same, since the definition of environments has changed. Specifically, the closure of a recursive function f is $\mathit{closure}(\mathit{mrule}, \eta)$, which has exactly the same form as normal functions, where η itself contains a reference to f. Unlike the previous assignments, **let rec** is not a syntactic sugar in TML.

Before starting to write code, you will have to complete the definition of the abstract machine K and design your own transition rules for TML as follows.

- Complete the definition of values v.
- Complete the definition of computation labels ϕ.
- Give the transition rules for the abstract machine K, including pattern matching.

TML is a subset of OCaml, and we adopt the relevant part of the operational semantics of OCaml, which we assume that you know exactly. This helps you design your own abstract machine K for TML. Now it's time to have fun!

4 Programming Instruction

Download the zip file `hw8.zip` from the course webpage or by running `receive` on the server `141.223.163.223`, and unzip it on your working directory.

First, see the file `store.ml` that implements a map from integer locations to `'a` values.

```
type 'a store
val empty : 'a store
val alloc : 'a -> 'a store -> int * 'a store
val deref : int -> 'a store -> 'a
val update : int -> 'a -> 'a store -> 'a store
```

The function `empty` returns an empty store $\cdot$; `alloc` v s stores a given value v in a fresh location l and returns the pair (l, s') with the updated store s'; `deref` l s fetches the value v stored in s at location l; and `update` l v s updates the store at location l with the given value v and returns the updated store s'. `Not_found` is raised if the location l is not available in s.

Second, see the file `env.ml` that implements a map from string identifiers to `'a` locations.

```
type 'a env
val empty : 'a env
val lookup : string -> 'a env -> 'a
val insert : string -> 'a -> 'a env -> 'a env
val singleton : string -> 'a -> 'a env
```

The function `empty` returns an empty environment $\cdot$; `lookup` x env returns the location related to x in env, and raises `Not_found` if $x \notin dom(env)$; `insert` x l env returns a environment with the same set of bindings as env plus a new binding $x \hookrightarrow l$; and `singleton` x l returns an environment with a single binding $x \hookrightarrow l$.

Next, see the file `hw8.ml`. The types `env`, `store`, and `config` correspond to the syntactic categories environment, store, and configuration, respectively, and are already defined in `hw8.ml`.

```
...
type location = int

type env = location Env.env

type value = CLOSURE of mrule * env
           | NOT_IMPLEMENT_VALUE

type store = value Store.store

type label = E of exp
           | V of value
           | NOT_IMPLEMENT_LABEL

type config = label list * env * store
...
let config2str _ = ""

let value2exp _ = raise NotImplemented

let step _ = raise NotImplemented
```

First, complete the definitions of types `value` and `label`, which correspond to the syntactic categories value and label, respectively. A closure value $\texttt{CLOSURE}(\mathit{mrule}, \eta)$, and an expression label $\texttt{E}(e)$ and a value label $\texttt{V}(v)$ are already given. The extra labels $\texttt{E}(e)$ and $\texttt{V}(v)$ are needed, because a computation is implemented as a list of labels of *the same type*. As a consequence, you may need auxiliary rules to transform $\texttt{E}(v)$ to $\texttt{V}(v)$ for *some* value v as follows:

$$\frac{\texttt{E}(v) \curvearrowright k}{\texttt{V}(v) \curvearrowright k} \; \langle\ \rangle_c$$

Second, implement the function `value2exp : value -> Tml.exp`. Given a primitive value v_p as defined below, `value2exp` returns a corresponding expression of type `Tml.exp`. For all other kinds of values, `value2exp` raises an exception `NotConvertible`.

$$\textsf{primitive value} \quad v_p \quad ::= \quad \mathit{num} \mid \textbf{true} \mid \textbf{false} \mid () \mid \mathit{op} \mid \mathit{cid} \mid \mathit{cid}\ v_p \mid (v_p, v_p)$$

Remark. **It is absolutely important to give a correct implementation of this function, since our test module assumes the correctness of your implementation of `value2exp`.** If you give a wrong implementation of `value2exp`, you will receive no credit for the entire assignment, because the test program will judge that your `step` function is faulty!

Third, implement the function `config2str`. There is no strict specification for this function, but you might find this function useful for debugging your code. Feel free to choose whatever specification you like, as we will not test this function. The default implementation returns just an empty string, and leave it as is if you want!

The final goal of this part is to implement the function `step : config -> config`, which takes a configuration of the abstract machine K, and returns the next configuration. It raises an exception `Stuck` if no progress can be made. We will test your implementation of `step` by examining the result v in the final configuration using *your* implementation of `value2exp`.

5 Testing and Submission

After implementing the function `step` in `hw8.ml`, run the command `ocamlbuild` to compile the sources files as follows to create an executable `main.native`:

```
$ ocamlbuild main.native
+ ocamlyacc parser.mly
19 shift/reduce conflicts.
Finished, 32 targets (0 cached) in 00:00:01.
```

Now you can run TML programs in the `example` directory (or your own TML program in a file) using the executable `main.native`. For example:

```
$ ./main.native example/fib.tml
21
$ ./main.native example/list.tml
(Cons (2, (Cons (1, Nil))))
```

You can use the command line argument `-debug` to print out the complete sequence of transitions using *your* implementation of `config2str`.

```
$ ./main.native example/test.tml -debug
...
```

Make sure that you can compile `hw8.ml` by running `ocamlbuild main.native`. When you have the file `hw8.ml` ready for submission, run the `handin` command in the same directory, and your file will be submitted automatically.