CS520 Programming Assignment 3

Posted: 17 Oct 2006
Due: 11:59pm, 9 Nov 2006

Overview The purposes of this assignment are:

1. Extend the language λ_t (in programming assignment 2) with effects, namely, *exceptions* and *references* to a language called λ_t^{eff}.
2. Implement a type-checker for λ_t^{eff}.
3. Implement an evaluator for λ_t^{eff}.

Syntax of λ_t^{eff} (extended from λ_t)

types	ty	$::=$	$\texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid ty_1 \rightarrow ty_2 \mid ty_1 * \ldots * ty_n$
			$\mid \texttt{ref}\ ty \mid \texttt{exn}$
contants	c	$::=$	$() \mid \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid \ldots$
operators	op	$::=$	$+ \mid - \mid * \mid / \mid \sim \mid \texttt{print} \mid \ldots$
terms	t	$::=$	$c \mid x \mid \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid op(t_1, \ldots, t_n) \mid \texttt{lam}\ (x : ty) \Rightarrow t$
			$\mid t_1(t_2) \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid \texttt{letrec}\ x : ty = t_1\ \texttt{in}\ t_2$
			$\mid (t_0, \ldots, t_n) \mid t.i \mid \texttt{fix}(t) \mid (t : ty)$
			$\mid \texttt{ref}\ t \mid t_1 := t_2 \mid !t \mid \texttt{raise}\ t$
			$\mid \texttt{try}\ t_0\ \texttt{with}\ cls$
patterns	pat	$::=$	$c \mid x$
clauses	cls	$::=$	$(pat_1 \Rightarrow t_1 \mid \ldots \mid pat_n \Rightarrow t_n)$

where a pattern can be either a constant or a variable.

Note that `unit` corresponds to `Unit`, the type constructor `ref` corresponds to `Ref` and () corresponds to unit (value), respectively, in Pierce's book.

Operators We assume the following operators in λ_t^{eff} with corresponding types:

$+$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{int}$	
$-$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{int}$	
$*$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{int}$	
$/$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{int}$	
$\sim$	:	$\texttt{int} \rightarrow \texttt{int}$	(* negation *)
$>$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$	
$>=$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$	
$<$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$	
$<=$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$	
$=$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$	
$<>$	:	$\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$	
`print`	:	$\texttt{string} \rightarrow \texttt{unit}$	

Abstract Syntax Definition of λ_t^{eff} in Ocaml

```
type stp =
    TpBase of string   (* base type *)
  | TpFun of stp * stp (* function type *)
  | TpTup of stp list  (* tuple type *)
  | TpExn (* exception type *)
  | TpRef of stp (* reference type *)

type ttm =
    TtmBool of bool    (* boolean constant *)
  | TtmInt of int      (* integer constant *)
  | TtmStr of string   (* string constant *)
  | TtmVar of string   (* variable *)
  | TtmIf of ttm * ttm * ttm   (* if-then-else term *)
  | TtmOp of string * ttm list (* built-in operator *)
  | TtmLam of string * stp * ttm (* lambda abstraction *)
  | TtmApp of ttm * ttm          (* application *)
  | TtmLet of string * ttm * ttm (* let-binding *)
  | TtmLetrec of string * stp * ttm * ttm (* letrec-binding *)
  | TtmTup of ttm list  (* tuple *)
  | TtmPro of ttm * int (* projection *)
  | TtmFix of ttm       (* fixed point *)
  | TtmAsc of ttm * stp (* ascription *)
  | TtmRef of ttm       (* reference *)
  | TtmLoc of int       (* location *)
  | TtmAssign of ttm * ttm (* assignment *)
  | TtmDeref of ttm     (* de-reference *)
  | TtmRaise of ttm     (* raise *)
  | TtmTry of ttm * (ttm * ttm) list (* try ... with ... *)
```

Static Semantics of λ_t^{eff}(rules similar as λ_t's are omitted)

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{ref}\ T_1}\ (\textbf{ty-ref})$$

$$\frac{\Gamma \vdash t_1 : \texttt{ref}\ T_1}{\Gamma \vdash\ !t_1 : T_1}\ (\textbf{ty-deref})$$

$$\frac{\Gamma \vdash t_1 : \texttt{ref}\ T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \texttt{unit}}\ (\textbf{ty-assign})$$

$$\frac{t_1 \in \{\text{constants, variables}\}}{\Gamma \vdash t_1 : \texttt{exn}}\ (\textbf{ty-exn})$$

$$\frac{\Gamma \vdash t_1 : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ t_1 : T}\ (\textbf{ty-raise})$$

$$\frac{\Gamma \vdash t_0 : T \quad \Gamma \vdash pat_1 : \texttt{exn} \quad \Gamma \vdash t_1 : T \quad \ldots \quad \Gamma \vdash pat_n : \texttt{exn} \quad \Gamma \vdash t_n : T}{\Gamma \vdash \texttt{try}\ t_0\ \texttt{with}\ pat_1 \Rightarrow t_1 \mid \ldots \mid pat_n \Rightarrow t_n : T}\ (\textbf{ty-try})$$

Note that all exceptions here are constants or variables, i.e. they do not carry values.
Remark: we do not need to maintain a store typing Σ in the rules as in Chapter 13 of Pierce's book because there are no locations in the source programs. Such a Σ is only needed if we are to type intermediary programs that mention locations.

Dynamic Semantics of λ_t^{eff}

We use l for *locations* and μ for *location store* and $\mu[l \mapsto v]$ to mean updating the store μ at location l by v.

$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{\texttt{ref } t_1 \mid \mu \ \rightarrow \ \texttt{ref } t_1' \mid \mu'} \ (\textbf{eval-ref1})$$

$$\frac{l \notin dom(\mu)}{\texttt{ref } v_1 \mid \mu \ \rightarrow \ l \mid \mu[l \mapsto v_1]} \ (\textbf{eval-refV})$$

$$\frac{t_1 \mid \mu \ \rightarrow \ t_1' \mid \mu'}{!t_1 \mid \mu \ \rightarrow \ !t_1' \mid \mu'} \ (\textbf{eval-deref1})$$

$$\frac{\mu(l) = v}{!l \mid \mu \ \rightarrow \ v \mid \mu} \ (\textbf{eval-derefL})$$

$$\frac{t_1 \mid \mu \ \rightarrow \ t_1' \mid \mu'}{t_1 := t_2 \mid \mu \ \rightarrow \ t_1' := t_2 \mid \mu'} \ (\textbf{eval-assign1})$$

$$\frac{t_2 \mid \mu \ \rightarrow \ t_2' \mid \mu'}{v_1 := t_2 \mid \mu \ \rightarrow \ v_1 := t_2' \mid \mu'} \ (\textbf{eval-assign2})$$

$$\frac{}{l := v \mid \mu \ \rightarrow \ () \mid \mu[l \mapsto v]} \ (\textbf{eval-assignL})$$

$$\frac{}{(\texttt{raise } v)t_2 \mid \mu \ \rightarrow \ \texttt{raise } v \mid \mu} \ (\textbf{eval-raise1})$$

$$\frac{}{v_1(\texttt{raise } v) \mid \mu \ \rightarrow \ \texttt{raise } v \mid \mu} \ (\textbf{eval-raise2})$$

Rules for `raise` v occuring in other contexts (`let`, *op*, etc.) are similar.

$$\frac{t_0 \mid \mu \ \rightarrow \ t_0' \mid \mu'}{\texttt{try } t_0 \texttt{ with } clauses \mid \mu \ \rightarrow \ \texttt{try } t_0' \texttt{ with } clauses \mid \mu'} \ (\textbf{eval-try1})$$

$$\frac{}{\texttt{try } v_0 \texttt{ with } cls \mid \mu \ \rightarrow \ v_0 \mid \mu} \ (\textbf{eval-tryV})$$

$$\frac{pat_i \neq pat_k \text{ for } k = 1, \ldots, i-1.}{\texttt{try } pat_i \texttt{ with } (pat_1 \Rightarrow t_1 \mid \ \ldots \ \mid pat_n \Rightarrow t_n) \mid \mu \ \rightarrow \ t_i \mid \mu} \ (\textbf{eval-tryExn})$$

Problem 1 (40pts): Based on the given static semantics, implement a function called `typecheck` in Ocaml which performs type checking for a λ_t^{eff} term. The `typecheck` function should be assigned the following type in Ocaml:

`typecheck : ttm` $\rightarrow$ `stp option`

Note that for a well-typed term t of type T, `typecheck`(t) should return `Some`(T); Otherwise, return `None`.

Problem 2 (50pts): Based on the given dynamic semantics, implement a function called `eval` in Ocaml which *evaluates closed well-typed* λ_t^{eff} *terms through the call-by-value strategy (you can adapt the small-step semantics to big-step semantics for efficiency).* The `eval` function should have type

$$\texttt{eval} : \texttt{ttm} \rightarrow \texttt{ttm}$$

in Ocaml. Note that *locations* are represented as natural numbers and you need to implement the *location store* (and operations on it) by yourself.

Implementation notes A few files (in **prog3.tar.gz**) are provided to start the assignment. You need to provide the actual implementations of the above functions based on the given code. Once all the code are ready, type **make** under the directory. If no error reported, an executable file called **evaluator** will be produced. You can test your code by typing

```
./evaluator filename
```

where *filename* should be replaced by some actual file path. There are some test cases provided in the **test** directory.

Grading The grading of the assignment is based on whether the required functionalities are correctly implemented. Please make sure your code can be compiled and tested on **csa2** because all submissions will be tested on **csa2**. There are **10pts** for

1. if the code is well organized.
2. if errors are properly handled.
3. if the code has necessary comments.
4. etc.