CS301
Session 9

Agenda

- The uScheme interpreter
- Approaches to solving the homework problem

Abstract syntax: top level

```
datatype toplevel = EXP    of exp
                  | DEFINE of name * lambda
                  | VAL    of name * exp
                  | USE    of name
```


Abstract syntax: expressions

```
datatype
exp = LITERAL of value
        | VAR     of name
        | SET     of name * exp
        | IFX     of exp * exp * exp
        | WHILEX  of exp * exp
        | BEGIN   of exp list
        | LETX    of let_kind *
                     (name * exp) list * exp
        | LAMBDA  of lambda
        | APPLY   of exp * exp list
and let_kind = LET | LETREC | LETSTAR
```

Abstract syntax: values

```
and value =
          NIL
        | BOOL      of bool
        | NUM       of int
        | SYM       of name
        | PAIR      of value * value
        | CLOSURE   of lambda * value ref env
        | PRIMITIVE of primitive
withtype primitive = value list -> value
     and lambda    = name list * exp
```


Design choices

- ML has all the features of uScheme (and then some)
- ...so interpreting uScheme in ML is pretty easy
- Choice: when do we represent a uScheme feature *directly* by the same feature in ML?
- Example: the store

Environment and store

- ML types for generic environments

```
type name = string
type 'a env = (name * 'a) list
```

- Instantiate to get a mapping from names to locations:

```
value ref env
```

- We're using the ML store to represent the uScheme store!

Manipulating environments

```
(* lookup and assignment of existing bindings *)
exception NotFound of name
fun find (name, []) = raise NotFound name
  | find (name, (n, v)::tail) = if name = n then v
else find(name, tail)

(* adding new bindings *)
exception BindListLength
fun bind(name, v, rho) = (name, v) :: rho
fun bindList(n::vars, v::vals, rho) = bindList
(vars, vals, bind(n, v, rho))
  | bindList([], [], rho) = rho
  | bindList _ = raise BindListLength
```

Initial environment

```
val emptyEnv = []

fun initialEnv() =
  let val rho = foldl
    (fn ((name, prim), rho)
     => bind(name, ref (PRIMITIVE prim), rho))
    emptyEnv (("+", arithOp op +  ) ::
              ("-", arithOp op -  ) ::
              ...
```


The function `eval`

```
fun eval(e, rho) =
  let fun ev(LITERAL n) = n
        | ev(VAR v) = ...
        | ev(SET (n, e)) = ...
        | ev(IFX (e1, e2, e3)) = ...
        | ev(WHILEX (guard, body)) = ...
        | ev(BEGIN es) = ...
        | ev(LAMBDA l) = ...
        | ev(APPLY (f, args))  = ...
        | ev(LETX (LET, bs, body)) = ...
        | ev(LETX (LETSTAR, bs, body)) = ...
        | ev(LETX (LETREC, bs, body)) = ...
  in  ev e
  end
```

Why is there no parameter "sigma"? Note the handling of "rho"!

Translating semantics to code

Variables

- Lookup

$$\frac{x \in \text{dom}\,\rho \qquad \rho(x) \in \text{dom}\,\sigma}{\langle \text{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle}$$

```
| ev(VAR v) = !(find(v, rho))
```

- What's that `!` doing there? Where does `rho` come from?

Assignment

- $$\frac{x \in \text{dom}\,\rho \quad \rho(x) = l \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{l \mapsto v\} \rangle}$$

```
| ev(SET (n, e)) =
    let val v = ev e
    in  find (n, rho) := v;
        v
    end
```

- What does `:=` mean? What's the reason for `;` ?

Let-binding

$$\frac{\begin{array}{c} l_1, \ldots, l_n \notin \text{dom}\,\sigma \\ \langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}, \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

```
| ev(LETX (LET, bs, body)) =
    let val (names, values) = ListPair.unzip bs
    in  eval (body, bindList(names, map (ref o ev) values, rho))
    end
```

- Where do the fresh locations come from? Notice the environment used for `eval`!

Let* binding

$$\frac{\begin{array}{c} l_1, \ldots, l_n \notin \text{dom}\,\sigma \\ \langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad \rho_1 = \rho\{x_1 \mapsto l_1\} \quad \sigma_1 = \sigma'\{l_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma'_{n-1} \rangle \quad \rho_n = \rho_{n-1}\{x_n \mapsto l_n\} \quad \sigma_n = \sigma'_{n-1}\{l_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

```
| ev(LETX (LETSTAR, bs, body)) =
    let fun step ((n, e), rho) = bind(n, ref (eval(e, rho)), rho)
    in  eval (body, foldl step rho bs)
    end
```

- Where is `rho` bound in the second line? In the third line? How is the sequencing controlled?

Letrec binding

- Left as an exercise for the reader

Evaluating functions

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle\langle\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle\rangle, \sigma \rangle}$$

```
| ev(LAMBDA l) = CLOSURE(l, rho)
```

- Remember:

```
datatype value = NIL
         ...
             | CLOSURE of lambda * value ref env
```

- This is the main key point for the assignment. What's the current type of the result of evaluating a lambda? What do you want to change it to?

Applying primitive functions

```
| ev(APPLY (f, args))  =
     (case ev f
        of PRIMITIVE prim => prim (map ev args)
```

- Another crucial point for the assignment: if lambda expressions are going to evaluate to a value of the same type as primitive functions, how should we expect to apply them to arguments?

Applying defined functions

$$\frac{\begin{array}{c} l_1, \ldots, l_n \notin \text{dom } \sigma \\ \langle e, \rho, \sigma \rangle \Downarrow \langle\langle\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c \rangle\rangle, \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}, \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

```
| ev(APPLY (f, args))  = (case ev f of ...
 | CLOSURE clo    =>
     let val ((formals, body), savedrho) = clo
         val actuals = map ev args
     in  eval(body, bindList(formals, map ref
                              actuals, savedrho))
       handle BindListLength => raise RuntimeError (...)
     end
```


Changing it

- What will happen to the code for applying user functions in your new version of the interpreter?
- How will you make sure the function body is evaluated in the correct environment?
- How will you test to make sure you did the right thing?

Approaching the assignment

- Big clue: "rename PRIMITIVE to PROCEDURE"
- Why? because now all procedures will have the same representation.
- A useful sidetrack: understand exactly how ML functions +, - etc. are used to construct the uScheme primitives.

Some clues

- The design choice being explored has to do with which uScheme features are to be represented by the corresponding ML features
- For a complete answer, pay attention to error handling; but in the beginning, ignore it
- A slight emendation: the result of the eval of a lambda has *the same type* as a primitive function

More clues

- Virtually all the code you need is already there in the interpreter. You just have to reorganize.
- It's a lot of work to do this the wrong way. It's very little work to do it the right way. Remember, the change is described as a "simplification"!

CS301
Session 10

Agenda

- Introduction to type systems
- A monomorphic type system for Impcore
- Types for arrays

Static vs. dynamic checking

- Dynamic checking in uScheme:

```
-> (define appendfoo (l) (append 'foo l))
appendfoo
-> (appendfoo '(1 2))
error: car applied to non-pair foo in (car l1)
```

- Static checking in ML:

```
- fun appendfoo l = "foo" @ l;
! Toplevel input:
! fun appendfoo l = "foo" @ l;
!                   ^^^^^
! Type clash: expression of type
!   string
! cannot have type
!   'a list
```


Checking and interpreters

- Dynamic type checks: integrated with evaluation - a single-stage interpreter
- Static type checks: first phase of a two-stage interpreter

Static type checking: why?

- Not just to annoy novice programmers
- Support for serious programming
  - Catch mistakes at compile time and reduce dependency on completeness of testing
  - Document the intended behavior of programs
  - Define interfaces between modules
- Support for optimizing compilers

Where we're going

- Typed Impcore: a simple - and annoying - type system
- Typed uScheme: a powerful - and still annoying - type system ...
- ... that motivates *type inference* as implemented for ML, Haskell, etc.

What Impcore types do

- In a well-typed Impcore program we know:
  - Every function (including primitives) receives the right number and type of actual parameters
  - Only Booleans are used for flow control (if and while)
- ...and we know this without ever running the program!

What they don't do

- We don't know if
  - there is division by 0
  - application of car or cdr to the empty list
  - illegal array indexing
  - infinite looping or recursion
  - wrong answers

Extending Impcore

- Values: integers, Booleans, arrays
- Add typing rules and a static type checker
- Show type soundness: if a well-typed expression *e* evaluates to *v*, then *v* is compatible with the type of *e*
  - `int` yields a number
  - `bool` yields 0 or 1 (!)
  - etc.

Extending the syntax

- Argument and result types must be declared, e.g.

```
(define int exp ((int b) (int e))
  (if (= 0 e) 1
      (* b (exp b (- e 1)))))
```


Extending the syntax (2)

- Array syntax example:

```
-> (define (array int) nzeroes ((int n))
       (array-make n 0))
nzeroes
-> (nzeroes 10)
[0 0 0 0 0 0 0 0 0 0]
```


The types of tImpcore

- Base types

```
int
bool
unit
```

- Array types

`(array` *type*`)`

Booleans

- Absence of Boolean literals leads to some oddities:

```
-> (= 0 0)
1
-> (= 0 1)
0
-> (if 1 2 3)
type error: Condition in if expression has type
int, which should be bool
-> (if (= 0 0) 2 3)
2
-> (val true (= 0 0))
1
-> (if true 2 3)
2
```


Reminder: Impcore semantics

- Remember that we needed three environments for Impcore: ξ, ϕ, ρ for global variables, functions, and formal parameters
- In our type system (sometimes called a *static semantics*) we have three type environments $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ to record the types of global variables, functions, and parameters

Type system

- Simple types: $\tau \;\; = \text{INT}|\text{BOOL}|\text{UNIT}|\text{ARRAY}(\tau)$
- Function types: $\tau_f = \tau_1 \times \ldots \times \tau_n \to \tau$
- Typing judgment for expressions: $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$
- Typing judgement for top-level items:

$$\langle t, \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma'_\phi \rangle$$

- Properties: deterministic, sound w.r.t. evaluation

Literals and variables

- Literals are numbers

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LITERAL}(v) : \text{INT}}$$

- Variable types are kept in the type environments

$$\frac{x \in \text{dom } \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\rho(x)}$$

$$\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\xi(x)}$$

Assignments

- To parameters

$$\frac{x \in \text{dom}\ \Gamma_\rho \quad \Gamma_\rho(x) = \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau}$$

- To globals

$$\frac{x \notin \text{dom}\ \Gamma_\rho \quad x \in \text{dom}\ \Gamma_\xi \quad \Gamma_\xi(x) = \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau}$$

Typing an if-expression

- The two arms must have the same type - why?

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{IF}(e_1, e_2, e_3) : \tau}$$

Typing a while-loop

- The value returned is the uninteresting "unit"

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}}$$

Typing a sequence

- Types 1 ... n-1 are uninteresting

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \ldots \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n}$$

Function application

- Using the function type environment

$$\frac{\Gamma_\phi(f) = \tau_1 \times \ldots \times \tau_n \rightarrow \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(f, e_1, \ldots, e_n) : \tau}$$

- We check that the actual parameters have the required types

Function definition

- Extending the function type environment

$$\frac{\Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \ldots \times \tau_n \rightarrow \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau), \Gamma_\xi, \Gamma_\phi, \rightarrow \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \ldots \times \tau_n \rightarrow \tau\}}$$

- Notice how we *assume* the formals have the correct types while we are typing the body!

Top level value binding

- We extend the global type environment

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle}$$

Typing arrays

- Introduction rule

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{INT} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-MAKE}(e_1, e_2) : \text{ARRAY}(\tau)}$$

- One of the elimination rules:

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \text{INT}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-GET}(e_1, e_2) : \tau}$$

A type-checking interpreter

- Instead of a read-eval-print loop we have a read-check-eval-print loop
- The type checking stage is a straightforward translation of the inference rules
- Now we see the usefulness of the polymorphic environment type from the uScheme interpreter:

```
typeof : exp * ty env * funty env * ty env -> ty
```


Abstract syntax

- Types and expressions

```
datatype ty    = INTTY | BOOLTY | UNITTY | ARRAYTY of ty
datatype funty = FUNTY of ty list * ty
datatype exp   = LITERAL of value
               | VAR     of name
               | SET     of name * exp
               | IFX     of exp * exp * exp
               | WHILEX  of exp * exp
               | BEGIN   of exp list
               | APPLY   of name * exp list
               | AGET  of exp * exp
               | ASET  of exp * exp * exp
               | AMAKE of exp * exp
               | ALEN  of exp
```


Abstract syntax

- Top level

```
  type userfun =
{ formals : (name * ty) list, body : exp, returns : ty }

  datatype toplevel = EXP    of exp
                    | DEFINE of name * userfun
                    | VAL    of name * exp
                    | USE    of name
```


Typing a function definition

$$\frac{\Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \ldots \times \tau_n \to \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\langle \textsc{define}(f, (\langle x_1 : \tau_1, \ldots, x_n : \tau_n\rangle, e : \tau), \Gamma_\xi, \Gamma_\phi, \to \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \ldots \times \tau_n \to \tau\}}$$

```
fun topty (t, globals, functions) =
    case t
...
  | DEFINE (name, {returns, formals, body}) =>
     let val (fnames, ftys) =
               ListPair.unzip formals
         val functions' =
               bind(name, FUNTY (ftys, returns), functions)
         val tau =
               typeof (body, globals, functions',
                        bindList(fnames, ftys, emptyEnv))
     in  if tau = returns then
           (globals, functions')
         else
           raise TypeError (...)
     end
```