CSE216 Programming Abstractions

Type Systems

YoungMin Kwon

Why Types

- Types provide implicit context
  - a + b: integer addition or floating point addition
  - new: allocates memory and calls proper constructor
- Types limit the set of permitted operations
  - Reduce mistakes in programming
    - E.g.: Prevent adding a number to a Boolean (1 + true)
- With types, programs are easier to read

```
(*I wish I knew the types of delay, curr, and stream…*)
let rec after_delay delay curr stream =
```

- Help compilers optimizing performances

Type Systems

- A type system consists of
  - Mechanism to
    - Define types
    - Associate them with language constructs
  - A set of rules for
    - Type equivalence
    - Type compatibility
    - Type inference

Type Checking

- Type checking
    - Process of ensuring that a program obeys the language’s type compatibility rules

- Strongly typed language
    - Prohibits the application of any unsupported operations to objects

    - Static type checking: type checking is performed at compile time

    - Dynamic type checking: type checking is performed at run-time

What is Type

- Denotational point of view
    - A set of values known as a domain
        - E.g.: {1, 2, 3, …}, {‘a’, ‘b’, ‘c’, …}, {true, false},…
    - Types are domains and the meaning of an expression is a value from the domain
    - Programmers may think user defined types as mathematical operations on sets
        - E.g.: Cartesian products for tuples: int * int as $\mathbb{Z}\times\mathbb{Z}$

What is Type

- Structural point of view
  - A type is either
    - A primitive type (int, char, boolean, …) or
    - A composite type (tuple, record, list…)
  - Programmers may think in terms of the way it is built from simpler types

What is Type

- Abstraction-based point of view
  - A type is an interface consisting of a set of operations
  - Programmers may think in terms of its meaning or purpose

Polymorphism

- Parametric polymorphism
  - Code takes a type as a parameter
  - Generics or templates in Java, C++
- Subtype polymorphism
  - A code designed to work with type T also works with T's subtypes
  - Most object oriented languages
- Combination of subtype and parametric polymorphism
  - Useful for containers: List<T> or Stack<T>

Type Checking

- Type equivalence
  - Whether two types are the same
- Type compatibility
  - When an object of a type can be used in a certain context
  - Type conversion, type coercion, non-converting type cast
- Type inference
  - Given the types of the subexpressions, find the type of the expression as a whole?

Type Equivalence

- Type equivalence
    - Whether two types are the same
- Structural equivalence
    - Two types are the same if they consists of the same components in the same way
    - E.g.: Algol-68, Modula-3, C, ML
- Name equivalence
    - Lexical occurrence of type definitions
    - E.g.: Java, C#

Type Equivalence

```
type R1 = record
    a, b: integer
end
```

```
type R2 = record
    a: integer;
    b: integer
end
```

```
type R3 = record
    b: integer;
    a: integer
end
```

- Example
  - In many languages R1 and R2 are structurally equivalent
  - In many languages R3 is not equivalent to R1 or R2.

Type Equivalence

```
type student = record
     name, address: string
     id: integer
end

x: student;
y: school;
…
x := y
```

```
type school = record
     name, address: string
     id: integer
end
```

- Problem with structural equivalence
  - Cannot distinguish types that the programmer may think of as different

- Name equivalence
  - If the programmer distinguishes the types, they are probably meant to be different

Type Conversion and Casts

- In a program, values of specific types are expected

```
a := expression
```

expression should have the same type as a

```
a + b
```

a and b are both integers or they are both floats

```
foo(v1: type1, v2: type2)
…
foo(expr1, expr2)
```

expr1 should be type1 and expr2 should be type2

Type Conversion and Casts

- Type conversion cases
  - Types are structurally equivalent, but the language uses name equivalence
    - Conversion is purely conceptual
  - Types have different sets of values, but the intersecting values are represented in the same way
    - `signed int` ↔ `unsigned int`
    - Runtime check: If the current value is in the intersect, use it (1 → 1). If not, runtime error (-1 → ?).

Type Conversion and Casts

- Type conversion cases (Cont’d)
  - Types have different representations, but some correspondence can be defined among their values
    - `int` ↔ `float`
    - Machine instructions for the conversion

Non-converting Type Casts

- Particularly, in systems programming
  - Change the type of a value without changing the underlying implementation
  - E.g. `malloc`
    - Represent heap as a large array of bytes
    - Reinterpret portions of the memory as pointers and integers

Type Compatibility

- Most language requires type compatibility rather than type equality
  - a + b: a and b must be compatible with some type that supports addition
    - E.g. 1 + 0.2
  - foo(a, b): a and b must be compatible with the formal parameters (subtypes)

Type Compatibility

- Type compatibility varies from language to language
  - E.g. in Ada, type S is compatible with type T if
    - S and T are equivalent
    - One is a subtype of the other or both are subtypes of the same base type
    - Both are arrays with the same number and type of elements

Coercion

- Type coercion
  - When necessary, a language performs an automatic, implicit conversion to the expected type
- Coercion is a controversial subject
  - Type conversion without programmer's explicit cast → it can weaken type security
  - Natural way to support abstraction and extensibility → easy to use new types with existing ones

Type Checking in Tiny

```
let str_max =
    "(lambda (x:num y:num)
             (if (>= x y) x y))"

let str_inside =
    "(lambda (lb:num ub:num x:num)
         (and (>= x lb) (>= ub x)))"

let str_cons =
    "(lambda (x:num
              y:num
              z:bool)
         (if z x y))"

let str_car =
    "(lambda (x:(bool -> num))
         (x true))"

let str_cdr =
    "(lambda (x:(bool -> num))
         (x false))"
```

Function parameters are decorated with types

Type Checking in Tiny

```
type kind
    = Boolean
    | Number
    | Function of kind * kind
    | Error

type expr
    = NUM  of int    (*number*)
    | BOOL of bool   (*Boolean*)
    | VAR  of string (*variable*)
    (*arithmetic exprs*)
    | ADD of expr * expr | SUB of expr * expr
    (*comparators*)
    | EQ of expr * expr  | GE of expr * expr
    (*logical exprs*)
    | AND of expr * expr | OR  of expr * expr | NOT of expr
    (*conditional expr*)
    | IF  of expr * expr * expr
    (*function definition: parameter, body*)
    | FUN of (string * kind) * expr
    (*function application: operator, operand*)
    | APP of expr * expr
```

SUB (NUM 1, BOOL true) is a valid expr, but it is an unsupported opr.

```
(*kind of expr in env*)
let rec kind expr env =

    (*look up the kind of a variable from an environment*)
    let rec lookup name env =
        match env with
        | [] -> Error
        | (n, knd)::rest -> if name = n
                            then knd
                            else lookup name rest in

    match expr with
    | BOOL b -> Boolean
    | NUM n  -> Number
    | VAR v  -> lookup v env

…
```

```
…
        | ADD (a, b) | SUB (a, b)  ->
            kind a env |> fun ka ->
            kind b env |> fun kb ->
            if ka = Number && kb = Number
            then Number
            else Error
        | EQ (a, b) | GE (a, b) ->
            kind a env |> fun ka ->
            kind b env |> fun kb ->
            if ka = Number && kb = Number
            then Boolean
            else Error
        | AND (a, b) | OR (a, b) ->
            kind a env |> fun ka ->
            kind b env |> fun kb ->
            if ka = Boolean && kb = Boolean
            then Boolean
            else Error
        | NOT a ->
            kind a env |> fun ka ->
            if ka = Boolean
            then Boolean
            else Error
…
```

```
| IF (cond, t_exp, f_exp) ->
    kind cond env  |> fun kc ->
    kind t_exp env |> fun kt ->
    kind f_exp env |> fun kf ->
    (*kind of t_exp and f_exp must match*)
    if kc = Boolean && kt = kf
    then kt
    else Error
```

```
| FUN ((param, k_param), body) ->
    (*find the kind of body in the extended env*)
    kind body ((param, k_param)::env) |> fun k_body ->
    if k_body != Error
    then Function (k_param, k_body)
    else Error

| APP (f, a) ->
    kind f env |> fun kf ->
    kind a env |> fun ka ->
    (match kf with
    | Function (kf_param, kf_ret) ->
        (*parameter kind of f and kind of a must match*)
        if kf_param = ka && ka != Error
        then kf_ret
        else Error
    | _ -> Error)
```

Type Inference

- Type inference
  - Determining the type of an expression

  - Examples
    - The result of an arithmetic operator usually has the same type as the operand
    - The result of a comparison is Boolean
    - The result of a function call is the type of the function body
    - The result of an assignment has the same type as the left-side

HM Type Inference Algorithm

HM: Hindley and Milner

- Type inference example

```
# let inc = fun x -> (+) 1 x;;
val inc : int -> int = <fun>
```

- Step 1: assign preliminary types
  - Assign type variables to subexpressions

Subexpression	Preliminary type
`fun x -> (+) 1 x`	A *(fun def)*
`x`	B *(formal param)*
`(+) 1 x`	C *(fun body, fun app)*
`(+) 1`	D *(fun, fun app)*
`(+)`	E *(fun)*
`1`	F *(actual param)*
`x`	G *(actual param)*

HM Type Inference Algorithm

- Step 2: collect type constraints

fun def

Subexpression	Preliminary type	Constraints
`fun x -> (+) 1 x`	A	A = B -> C
`x`	B	
`(+) 1 x`	C	D = G -> C
`(+) 1`	D	E = F -> D
`(+)`	E	E = int -> (int -> int)
`1`	F	F = int
`x`	G	G = B

fun app

HM Type Inference Algorithm

- Step 3: solve type constraints
  - Find a type assignment to type variables that can satisfy all type constraints

- `F = int`
- `D = int -> int`
- `G = int, C = int`
- `B = int`
- `A = int -> int`

Constraints
`A = B -> C`
`D = G -> C`
`E = F -> D`
`E = int -> (int -> int)`
`F = int`
`G = B`

HM Type Inference Algorithm

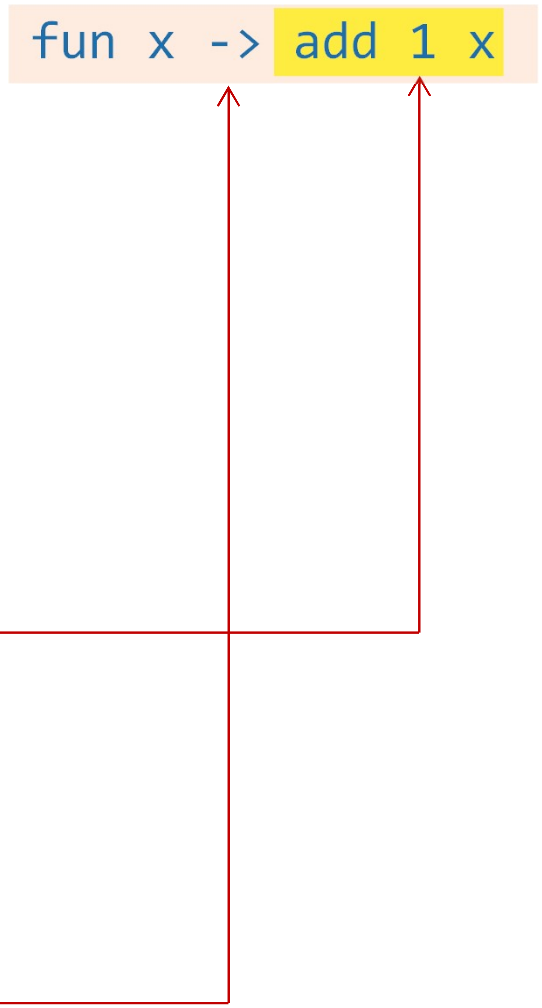
- Type Constraint collection
  - Assign fresh type variables to
    - Each function parameter
      ⇒ Let D(x) be the type variable for a function parameter x
    - Each subexpression of an expression
      ⇒ Let U(e) be the type variable for a subexpression e

Type Constraints

`fun x -> add 1 x`

- Generate type constraints
- For a constant c
  - `U(c) = type of c`
  - E.g.
    - Let U(1) be A, then A = int
    - Let U(add) be A, then A = int → int → int
- For a variable x
  - `U(x) = D(x)`
  - E.g
    - Let D(x) be A, U(x) be B, then A = B

Type Constraints


`fun x -> add 1 x`

- For a conditional expr if c then e_t else e_f
  - U(c) = bool, U(e_t) = U(e_f), U(if c then e_t else e_f) = U(e_t)
  - E.g.
    - Let U(if x then y else z) be A, U(x) be B, U (y) be C, U (z) be D then B = bool, C = D, A = C
- For a function application e1 e2
  - `U(e1) = U(e2) → U(e1 e2)`
  - E.g.
    - Let U(add 1 x) be A, U(add 1) be B, U(x) be C, then B = C → A
- For a function definition fun x -> e
  - `U(fun x -> e) = D(x) → U(e)`
  - E.g.
    - Let U(fun x -> y) be A, D(x) be B, U(y) be C, then A = B → C

Type Inference in Tiny: Constraints

```
(*kind (type) expression
    e.g.: KB -> KN -> KV0
*)
type kexpr
    = KB
    | KN
    | KV of int (*kind variables like k0, k1, k2, ...*)
    | KF of kexpr * kexpr (*functions like k1 -> k2*)


(*fresh variable*)
let newvar () =
    let open StateMonad in
    get         >>= fun i ->
    set (i+1) >>= fun () -> ret (KV i)
```

state monad:
similar to |>,
function composition

state monad:
returns value in
monad type

```
(*constr: type constraints for kvar = kexpr
    kvar:  type variable for kexpr
    kexpr: expression
    env :  variable name to kind variable map
    return: list of constraints (kexpr = kexpr)...
*)
let rec constr kvar kexpr env =
    let open StateMonad in

    (*Look up the kind expr associated with name from an environment*)
    let rec lookup name env =
        match env with
        | [] -> Printf.printf "%s" name; assert false
        | (n, ke)::rest -> if name = n
                        then ke
                        else lookup name rest in

    match kexpr with
    | BOOL b -> ret [(kvar, KB)]
    | NUM n  -> ret [(kvar, KN)]
    | VAR v  -> ret [(kvar, lookup v env)]
```

state monad:
returns value in
monad type

state monad:
each newvar ()
returns a different
fresh variable

```
| ADD (a, b) | SUB (a, b) ->
    newvar ()        >>= fun kv1 ->
    newvar ()        >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KN)::(kv1, KN)::(kv2, KN)::c2@c1 |> ret
```

state monad:
similar to |>,
function composition

state monad:
returns value in
monad type

```
| EQ (a, b) | GE (a, b) ->
    newvar ()        >>= fun kv1 ->
    newvar ()        >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KB)::(kv1, KN)::(kv2, KN)::c2@c1 |> ret
```

append list

```
| AND (a, b) | OR (a, b) ->
    newvar ()        >>= fun kv1 ->
    newvar ()        >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KB)::(kv1, KB)::(kv2, KB)::c2@c1 |> ret
| NOT a ->
    newvar ()        >>= fun kv1 ->
    constr kv1 a env >>= fun c1 ->
    (kvar, KB)::(kv1, KB)::c1 |> ret
```

```
| IF (c, t, f) ->
    newvar ()        >>= fun kv1 ->
    newvar ()        >>= fun kv2 ->
    newvar ()        >>= fun kv3 ->
    constr kv1 c env >>= fun c1 ->
    constr kv2 t env >>= fun c2 ->
    constr kv3 f env >>= fun c3 ->
    (kvar, kv2)::(kv1, KB)::(kv2, kv3)::c3@c2@c1 |> ret
```

```
| FUN (v, e) ->        (*function definition*)
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    (v, kv1)::env      |>        (*extend env*)
    constr kv2 e       >>= fun c1 ->
    (kvar, KF (kv1, kv2))::c1 |> ret

| APP (f, a) ->        (*function application*)
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    constr kv1 f env >>= fun c1 ->
    constr kv2 a env >>= fun c2 ->
    (kv1, KF (kv2, kvar))::c2@c1 |> ret
```

Unification

- How to solve type constraints
  - Substitution: substitute a type variable in a type expr with an associated type expr

```
(*substitution that substitutes kvar in ke with kexp*)
let substitution kvar kexp =
    let rec iter ke =
        match ke with
        | KB -> KB
        | KN -> KN
        | KV _ -> if kvar = ke then kexp else ke
        | KF (ke1, ke2) -> KF (iter ke1, iter ke2) in
    iter in
```

  - A composition of substitutions is a substitution

```
fun ke -> ke |> subst1 |> subst2
```

Unification

- Unifier
  - A substitution U is a unifier of a constraint e1 = e2 if (U e1) = (U e2)
- HM type inference algorithm
  - Given an expression, generate a set C of type constraints
  - Find a unifier U that unifies all constraints in C

```
(*return a unifier (substitution) for the constraints in clist*)
let rec unifier clist =
  match clist with
  | [] -> fun x -> x (*id substitution: no substitution*)
  | hd::tl ->
    match hd with
    | (KV v, ke) | (ke, KV v) ->
      if contains (KV v) ke
      then assert false (*recursive def is not supported*)
      else
        let sub_h = substitution (KV v) ke in
        let sub_t = tl
            |> List.map (fun (a, b) -> (sub_h a, sub_h b))
            |> unifier in
        (*unifier: composed substitutions*)
        fun e -> e |> sub_h |> sub_t

    | (KF (a, b), KF (c, d)) -> (a, c)::(b, d)::tl |> unifier

    | (a, b) -> if a = b
          then unifier tl      (*hd is already unified*)
          else begin           (*cannot unify: type error*)
            print a; print b; assert false
          end in
```

Unification Example

unprocessed constraints

substitutions

```
A = B -> C
D = G -> C
E = F -> D
E = int -> (int -> int)
F = int
G = B
```

```
A : B -> C
D = G -> C
E = F -> D
E = int -> (int -> int)
F = int
G = B
```

```
A : B -> C
D : G -> C
E = F -> (G -> C)
E = int -> (int -> int)
F = int
G = B
```

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F -> (G -> C) = int -> (int -> int)
F = int
G = B
```

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F = int
G -> C = int -> int
F = int
G = B
```

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F : int
G -> C = int -> int
int = int
G = B
```

Unification Example

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F : int
G = int
C = int
int = int
G = B
```

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F : int
G : int
C = int
int = int
int = B
```

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F : int
G : int
C : int
int = int
int = B
```

unifier

```
A : B -> C
D : G -> C
E : F -> (G -> C)
F : int
G : int
C : int
B : int
```

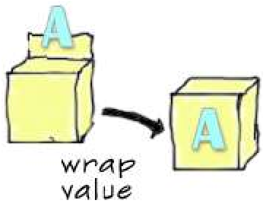
```
unifier A => B -> C
          => B -> int
          => int -> int
```

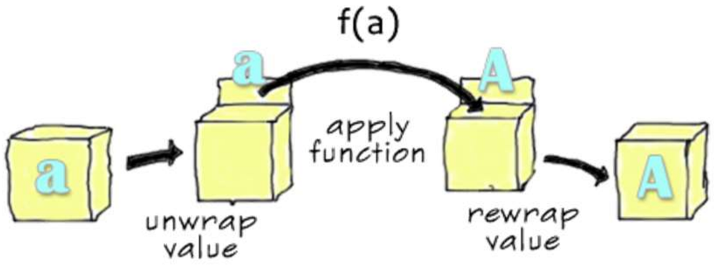
```
(*type_infer: infer the type of str_expr*)
let type_infer str_expr env =
    parse str_expr      |>  fun expr ->
    newvar ()          >>= fun kv ->
    constr kv expr env >>= fun clist ->
    unify clist         |>  fun unifier ->
    unifier kv          |>
    ret
(*define: extends env with the value of string expression*)
let define name str_expr env =
    type_infer str_expr env >>= fun kexp ->
    (name, kexp)::env        |>
    ret

let str_id =   "(lambda (x) x)"
let str_max =  "(lambda (x y) (if (>= x y) x y))"
let str_cons = "(lambda (x y sel) (if sel x y))"
let str_car =  "(lambda (x) (x true))"
let str_cdr =  "(lambda (x) (x false))“

let mon_def = []
    |>  define "id"   str_id
    >>= define "max"  str_max
    >>= define "cons" str_cons
    >>= define "car"  str_car
    >>= define "cdr"  str_cdr
```

```
mon_def >>= fun env ->                                expected results

type_infer "id" env          >>= fun k -> print k;    (k1) -> (k1)
type_infer "(id 1)" env      >>= fun k -> print k;    num
type_infer "(id true)" env   >>= fun k -> print k;    bool

type_infer "max" env         >>= fun k -> print k;    (num) -> ((num) -> (num))
type_infer "(max 1)" env     >>= fun k -> print k;    (num) -> (num)
type_infer "(max 1 2)" env   >>= fun k -> print k;    num

type_infer "cons" env        >>= fun k -> print k;    (k14) ->
                                                        ((k14) -> ((bool) -> (k14)))
type_infer "(cons 1)" env    >>= fun k -> print k;    (num) -> ((bool) -> (num))
type_infer "(cons 1 2)" env  >>= fun k -> print k;    (bool) -> (num)
type_infer "(cons true)" env >>= fun k -> print k;    (bool) -> ((bool) -> (bool))
type_infer "(cons true false)" env                    (bool) -> (bool)
                             >>= fun k -> print k;

type_infer "car" env         >>= fun k -> print k;    ((bool) -> (k25)) -> (k25)
type_infer "(car (cons 1 2))" env                     num
                             >>= fun k -> print k;

type_infer "cdr" env         >>= fun k -> print k;    ((bool) -> (k30)) -> (k30)
type_infer "(cdr (cons 1 2))" env                     num
                             >>= fun k -> print k;

ret ()
```

Optional: State Monad

```
(*state monad*)
module StateMonad = struct
    (*monad operations*)
    (*return: wraps values in a monad type*)
    let ret value =
        fun state -> (value, state)

    (*bind: composes functions that return a monad type*)
    let (>>=) mon f =
        fun state ->
            mon state |> fun (v, s) ->
            (f v) s

    (*other operations for state monad*)
    let get =
        fun state -> (state, state)

    let set state' =
        fun state -> ((), state')
end
```





Optional: State Monad

```
let nth_nat n = (*n-th natural number*)
    let open StateMonad in
    let nat () = (*natural numbers*)
        get       >>= fun i ->
        set (i+1) >>= fun () ->
        ret i in

    let rec iter n =
        if n = 0
        then nat ()
        else nat () >>= fun _ ->
             iter (n-1) in

    (iter n) 0 |> fun (v, s) -> v

let _ = nth_nat 10
```

```
module StateMonad = struct
    let ret value =
        fun state -> (value, state)

    let (>>=) mon f =
        fun state ->
            mon state |> fun (v, s) ->
            (f v) s

    let get =
        fun state -> (state, state)

    let set state' =
        fun state -> ((), state')
end
```

Optional: State Monad

```
let sum n =
    let open StateMonad in
    let nat () = (*natural numbers*)
        get       >>= fun i ->
        set (i+1) >>= fun () ->
        ret i in

    let rec iter n =
        if n = 0
        then nat ()
        else nat ()     >>= fun i ->
             iter (n-1) >>= fun j ->
             i + j |> ret in

    (iter n) 0 |> fun (v, s) -> v

let _ = sum 10
```

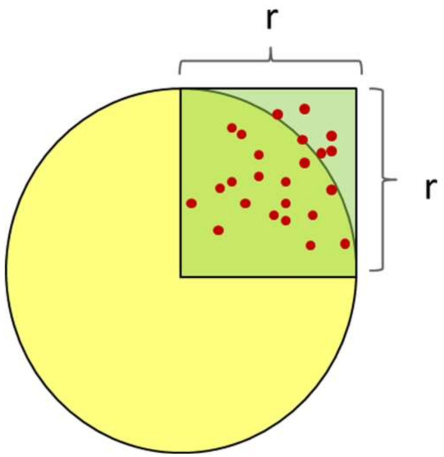
```
module StateMonad = struct
    let ret value =
        fun state -> (value, state)

    let (>>=) mon f =
        fun state ->
            mon state |> fun (v, s) ->
            (f v) s

    let get =
        fun state -> (state, state)

    let set state' =
        fun state -> ((), state')
end
```

Optional: State Monad

```
let fibo n =
    let open StateMonad in
    let fib () = (*fibonacci numbers*)
        get          >>= fun (i, j) ->
        set (i+j, i) >>= fun () ->
        ret i in

    let rec iter n =
        if n = 0
        then fib ()
        else fib () >>= fun _ ->
             iter (n-1) in

    iter n (1, 0) |> fun (v, s) -> v

let _ = fibo 5
```

```
module StateMonad = struct
    let ret value =
        fun state -> (value, state)

    let (>>=) mon f =
        fun state ->
            mon state |> fun (v, s) ->
            (f v) s

    let get =
        fun state -> (state, state)

    let set state' =
        fun state -> ((), state')
end
```

```
(*random number generator*)
let rand rmax =
    let open StateMonad in
    let next x = (x * 16807) mod 0x7fffffff in
    get    >>= fun x ->
    next x |>  fun r ->
    set r  >>= fun () -> ret (r mod rmax)

(*estimation of pi*)
let pi n =
    let open StateMonad in
    let radius = 1000 in
    let rseed  = 2    in
    let inside x y = x * x + y * y < radius * radius in
    let rec iter nr_in i =
        if i = n then
            ret (4. *. (float nr_in) /. (float n))
        else
            rand radius >>= fun x ->
            rand radius >>= fun y ->
            if inside x y
            then iter (nr_in + 1) (i + 1)
            else iter nr_in       (i + 1) in
    (iter 0 0) rseed |> fun (v, s) -> v

let _ = pi 100000
```



Assignment 8

- Tiny Type Checker in CPS
  - In this assignment, implement to_str and kind functions of Tiny Type Checker in CPS
  - Download tiny_type_check_cps.zip and implement to_str and kind functions in CPS
  - Submit type.ml to blackboard
- Due date: TBD

Assignment 8

```
#use "parser.ml"

(*print kind*)
let print_kind knd =
    (*TODO: implement to_str in CPS*)
    let rec to_str knd k =


    to_str knd (Printf.printf "%s\n")

(*kind of expr in env*)
(*TODO: implement kind in CPS*)
let rec kind expr env k =
    (*look up the kind of a variable from an environment*)
    let rec lookup name env =
        match env with
        | [] -> Error
        | (n, knd)::rest -> if name = n
                            then knd
                            else lookup name rest in
```

Assignment 8

```
(* expected results

-- Function Defs -------------
(num -> (num -> num))
(num -> (num -> (num -> bool)))
((num -> (num -> (num -> bool))) -> (num -> (num -> (num -> bool))))
(num -> (num -> (bool -> num)))
((bool -> num) -> num)
((bool -> num) -> num)
- : unit = ()
-- Function Apps -------------
num
bool
bool
Num
- : unit = ()

*)
```