CSE 341 Midterm – Taylor Blau

Slow down, double check...

1 Primitives

`int` An integer. Ex. `1`, `2`, etc.

`real` A real number. Ex. `1.0`, `Math.pi`, etc.

`string` A string of characters. Ex. `"CSE 341"`.

`char` A single character. Ex. `#"C"`.

`'a list` An list of other things that share the same type.

`'a option` Either `SOME 'a` or `NONE`.

`unit` The `unit` singleton: `()`.

2 Expressions

`val`-binding Introduces a binding: `val x = 5;`.

`fun`-binding Introduces a function binding (see: below).

`let`-expression Composes `val` and `fun` bindings.

`if`-expression Executes e_1, then e_2, otherwise e_3.

`case`-expression Branches on a "one-of" type (see: below).

3 Datatypes, records

Record values are values that have field names and values:

```
(* {f1: T1, ..., fn: TN} *)
val _ = {f1 = v1, ..., fn = vn}
val _ = (#f1 e1)
```

Datatype bindings have a type and one (or more) constructors:

```
datatype exp = Constant of int
             | Addition of exp * exp
             (* ... *)
```

and can be pattern-matched recursively:

```
fun eval exp =
  case exp of
      Constant i => i
    | Addition (e1, e2) = (eval e1) +
                          (eval e2)
    | _ = (* ... *)
```

The `type` keyword defines instead a *type synonym*, not a new type:

```
type cartesian = int * int
```

Since `type` introduces no new constructor bindings, this type may be used *interchangeably* with any place that expects or receives an `int * int`.

An *equality type* (denoted `''a`) means it is an unconstrained type α which defines equality against other α's.

3.1 Tail-recursion

Given a function defined as

```
fun factorial n =
  if n = 0 then 1
  else
    n * (factorial (n - 1))
```

Will take exponential stack space to compute, and instead can be tail-call optimized as:

```
fun factorial n =
  let fun aux (n, acc) =
    if n = 0
    then acc
    else aux(n-1, n*acc)
  in aux(n, 1)
  end
```

A function call is in the *tail-position* (and therefore, will be tail-call optimized) if:

1. If an expression is not in the tail position, then none of its sub-expressions are either.
2. `f` is the last function call in the enclosing expression.
3. If an `if`-expression is in the tail position, then both of its subexpressions are.

4 Exceptions

The `exception` binding creates a new exception type:

```
exception E1
exception E2 of int * int
```

The `raise` function raises an exception:

```
raise E1
raise (E2 (1, 2))
```

The `handle` expression rescues an exception:

```
(* ... *) handle E1 => (* ... *)
```

or fails to catch an exception, and the propagation continues up to and including termination of the program.

5 First-class Functions

Functions are values, and can be used in any other place that values can be used. Functions may be accepted as arguments, provided as return values, or etc.

Anonymous functions are defined as:

```
(fn (x) => ...)
```

5.1 Lexical scope

When a function is defined, it is evaluated to a closure. The closure composes the function definition, and the environment in which the function is evaluated. This environment is exactly the environment in which the function was *defined* extended with a reference to the function itself (if non-anonymous).

1. A function body is not evaluated until the function is called.
2. A function body is evaluated every time a function is called.
3. A variable binding evaluates its expression when the binding is evaluated, not every time it is used.

5.2 Composition

Functions can be composed using the `o` function:

```
val fn_1 = fn : 'a -> 'b
val fn_2 = fn : 'b -> 'c
val fn_3 = fn_2 o fn_1 (* 'a -> 'c *)
```

5.3 Currying

Functions can be tupled:

```
(* fn : int * int * int -> int *)
fun x (a, b, c) = a + b + c
```

or curried:

```
(* fn : int -> int -> int -> int *)
fun x a b c = a + b + c
```

6 References

1. The `ref` function creates a *new* reference to its argument.
2. The `!` function "de-references" the value inside the reference.
3. The `:=` function replaces the value inside the reference, and returns `()`.

7 Modules

Modules contain a list of bindings, and are namespaced under their top-level module name, in this case `M`.

```
structure M = struct bindings[...] end
```

Signatures can contain a list of expected bindings, and are *satisfied* by modules explicitly:

```
signature S = sig
  type t_abstract
  type t_concrete = int * int
  val my_fun : int -> int
  val my_val : int
end

structure M :> S = struct ... end
```

A signature `S` is matched by module `M` if the following hold:

1. Every non-abstract type in `S` is provided in `M` as given.
2. Every abstract type in `S` is provided in `M` in some way.
3. Every `val`-binding in `S` is provided in `M`, possibly with a more general or less abstract type.
4. Every `exn`-binding in `S` is provided in `M`.
5. Any additional bindings in `M` *not* specified in `S` are OK.

8 Mutual Recursion

To have two functions call one another, use the `and` keyword in a `fun`-binding:

```
fun expect_1 xs =
  case xs of
      1::xs' => expect_0 xs'
    | _ => false
and expect_0 xs =
  (* ... *)
```

9 Type Inference

To determine the type of a function, assume the following steps:

1. Determine the types of a binding-set in order.
2. Analyze all necessary facts.
3. Use type variables for unconstrained types.
4. Enforce value restriction.

10 Equivalence

Two functions are equivalent if they:

1. Produce equivalent results given equivalent inputs.
2. Exhibit the same (non-)termination behavior.
3. Mutate non-local memory similarly.
4. Perform the same input and output.
5. Raise the same exceptions.

There are a set of four standard equivalences:

1. Consistently rename bound variables and uses.
2. Use a helper function, or do not.
3. Perform unnecessary function wrapping, or do not.
4. ML `let`-bindings are syntatic sugar for function calls:

```
let val x = e1
in e2 end

(fn x => e2) e1
```

11 Standard library

11.1 List

`@ : 'a list * 'a list -> 'a list`

`map : ('a -> 'b) -> 'a list -> 'b list`

`filter : ('a -> bool) -> 'a list -> 'a list`

`foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

11.2 String

`implode : char list -> string`

`explode : string -> char list`

`sub : string * int -> char`

`size : string -> int`

12 Gotcha's

1. `int` vs. `real` usage.
2. Unnecessary function wrapping.
3. Currying vs. tupling.
4. Datatype bindings introduce constructors.

13 Racket Syntax

Atom e.g., `#t`, `#f`, `"hi"`, ...

Special Form e.g., `define`, `lambda`, macro definitions.

Sequence Of `(t1 t2 ... tn)`, if `t1` is a special form, semantics are special, otherwise function call.

13.1 `cond`

```
(cond [e1a e1b] [...])
```

13.2 Local Bindings

```
(let ([t1 e1] [...]) b)
```

`let` Expressions are evaluated without extending the environment.

`let*` Expressions are evaluated by extending the environment.

`letrec` All bindings are in environment, evaluated in order.

`define` Local definition.

13.3 `set!`

`set!` sets an identifier to a new expression, replacing its contents in locations that *have not* been evaluated yet.

13.4 Cells

`cons` makes a pair, where lists are sequences of pairs that end with `null`, and improper lists are those that do not.
`cons`-cells are immutable, but `mcons`-cells are not, and can be accessed with `mcons`, `mcar`, `mcdr`, `mpair?`, `set-mcar!`, and `set-mcdr!`.

13.5 Thunks

Thunks delay execution by not computing values until they are needed.

```
(define (thunk x) (lambda () x))
```

13.6 Promises

Thunks can make expensive computations more than once. Use a promise to keep track of this and only evaluate thunks once.

```
(define (delay th) (mcons #f th))
(define (force p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
             (set-mcdr! p ((mcdr p)))
             (mcdr p))))
```

13.7 Streams

Streams are an infinite sequence of values, and the stream creation is controlled by the consumer, much like `UNIX` pipes.

```
(* '(next-answer . next-thunk) *)
(define nats
  (letrec ([f (lambda (x)
    (cons x (lambda () (f (+ x 1)))))])
  (lambda () (f 1))))
```

13.8 Struct

```
(struct t (bar baz quux) #:transparent)
```

Introduces the following:
`(t e1 e2 e3)` A constructor for `t`. Evaluates each sub-expression.
`(t? e)` Evaluates `e` and returns whether or not it's a `t`.
`(t-field e)` Evaluates `e` and access the `field` field.

14 Implementing Languages

All languages are interpreted; dependent on what you mean by "interpreter". You can implement a language and skip the parse step by making the syntax itself an Abstract Syntax Tree (AST).

1. All expressions evaluate to other expressions.
2. All expressions are evaluated in an environment, earlier bindings are shadowed.
3. Higher-order functions evaluate to closures, `(struct closure (env fun))`, and calls extend the function with (1) a reference to itself, (2) a binding to the formal parameter.

The free variables of a function are those that are used in the function, but not introduced by the function.
Racket functions can produce ASTs that are themselves considered macros (without hygiene).

15 Soundness, completeness

Soundness A type system is *sound* if it never accepts a program that does "prevented" behavior, i.e., it produces no false negatives.

Completeness A type system is *complete* if it never rejects a program that will not do "prevented" behavior, i.e., it produces no false positives.

15.1 Claims

1. Dynamic/static is more convenient
2. Static prevents usefulness, lets you tag as needed.
3. Static catches bugs earlier, but they are only easy ones.
4. Static/dynamic is faster at runtime.
5. Easier code re-use with static/dynamic.
6. Static/dynamic better for prototyping/evolution.

16 Ruby

16.1 Class-based OOP

1. All values are object references.
2. Objects communicate via method calls.
3. Each object has private state.
4. Each object is an instance of a class.
5. A class determines object's behavior.

16.2 Class Syntax, Semantics

```
class Point # < Object
  ORIGIN = Point.new(0, 0)

  attr_reader :x, :y
  def initialize(x, y)
    @x, @y = x, y
  end

  # public (private, protected) ...
  def distance
    Math.sqrt(x*x + y*y)
  end
end
```

As above:

1. Initialization is done in `initialize`.
2. Class variables begin with `@@`, instance with `@`.
3. Class constants begin are called by `Point::ORIGIN`.
4. State is private, `p.x` is possible via `attr_reader`.
5. Methods are implicitly `public` (accessible everywhere), but can be `protected` (class, subclass), or `private` (object only).

16.3 Arrays

16.4 Blocks

Blocks are present throughout the standard library, each method implicitly takes one, and called with `yield`:

```
(1..10).each { |x| puts x }
(1..10).select { |x| x.even? }
(1..10).map { |x| x ** 2 }
(1..10).inject(0) { |x, a| a + x }
```

Blocks are second class, and cannot be passed, but the `lambda` method (or `Proc.new`) both take a block and stores them as objects with `#call`.

16.5 Sub-classing

Sub-classing allows you to override, refer to and add methods from a superclass. Sub-classing does not provide instance variables.
In the following, let `class C < D`:

`C.new.class == C`.

`C.new.class.superclass == D`.

`C.new.class.superclass.superclass == Object`.

`C.new.is_a?(D) == true`.

`C.new.instance_of?(D) == false`.

16.6 Dynamic Dispatch

In `e0.m(e1, ..., en)`:

1. Evaluate all expressions to `obj0`, `obj1`, ..., `objn`.
2. If `m` is defined in the class or mixins of `obj0`, use that with `self` bound to `obj0`.
3. Otherwise recur to the superclass.
4. Finally, call `#method_missing`.

16.6.1 Dynamic Dispatch in Racket

```
(struct obj (fields methods))
```

Where `fields` is a list of `mcons` cells, and `methods` is a list of `cons` cells, as:

```
(mcons 'x 17)
(cons 'get-x (lambda (self args)
  (get self 'x))
```

And helper functions are `assoc-m`, `(get obj fld)`, `(set obj fld v)`, `(send obj msg . args)`.

16.7 Double Dispatch

Double dispatch can be thought of the "behavior-grid" for a given method that takes two arguments. In ML, this can be expressed as a pattern-match with n^2 branches (or less with commutativity). In OOP, this can be expressed as double dispatch.

```
class Rock
  def battle(other)
    other.battle_rock(self)
  end
  def battle_rock(rock);          end
  def battle_paper(paper);        end
  def battle_scissors(scissors); end
end
```

Multimethods does this at run-time and is a bad fit for Ruby. Java has a lesser feature with static overloading.

16.8 Multiple Inheritance

Multiple inheritance causes problems when a "diamond" is formed in the sub-class tree. Mixins solve this problem and are a collection of methods. Notable mixins in Ruby are `Comparable`, `Enumerable`. Multiple interfaces do not have these problems.

17 Subtyping

Subtyping exhibits width, permutation, transitivity, and reflexivity.

Width subtyping If a record has more information, accept it as a subtype.

Depth subtyping Unsound with mutability.

Return types covariancy If `t2 <: t3`, `t1->t2 <: t1->t3`.

Argument contravariancy If `t3 <: t1`, `t2 <: t4`, then `t1->t2 <: t3->t4`.