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.