def func(x, y) → fun x y → λx. λy ← currying ex.

CS 425

Assignment 4(ML, Typechecking)

1 More ML

*Defn

Curry : ((a + b) → c) → (a → (b → c))

Uncurry : (a → (b → c)) → ((a + b) → c)

Type

1.1 Currying

Mitchell Problem 5.6

```
fun curry f = fn x => fn y => f(x,y)
fun uncurry f = fn(x,y) => f x y
```

fun

~~1.2 Disjoint Unions~~

~~Mitchell Problem 5.7~~

```
fun generate integers i = Cons(i, () => generate integers(i+1))
```

1.3 Lazy Evaluation and Functions

Mitchell Problem 5.8 (a) and (b)

```
val list = generate integers (0)
    list = cons(1, func)
```

2 Typechecker

You will understand and complete a type checker for a *statically typed language*. We assume function types are provided by the user (this is the approach of languages like C++ and Java).

Here is the syntax of the language we will work with:

```
e ::= x | n | true | false | iszero | succ | pred
    | if e then e else e | fn x : t => e | e e | (e)
t ::= 'a | int | bool | t -> t
```

Here 'a is a *type variable*, used for polymorphic types. As in SML, we'll assume that → associates to the right.

In the above grammar, x stands for an identifier; n stands for a non-negative integer literal; true and false are the boolean literals; succ and pred are unary functions that add 1 and subtract 1 from their input, respectively; iszero is a unary function that returns true if its argument is 0 and false otherwise; if e1 then e2 else e3 is a conditional expression; $fn\ x : t \Rightarrow e$ is a function with parameter x and body e; e e is a function application; (e) is to control parsing. It should be clear to you that the above grammar is quite ambiguous. For example, should $fn\ f : t \Rightarrow f\ f$ be parsed as $fn\ f : t \Rightarrow (f\ f)$ or as $(fn\ f : t \Rightarrow f)\ f$? We can resolve such ambiguities by adopting the following conventions (which are the same as in SML):

- Function application associates to the left. For example, e f g is (e f) g, not e (f g).
- Function application binds tighter than if, and fn. For example, $fn\ f : t \Rightarrow f\ 0$ is $fn\ f : t \Rightarrow (f\ 0)$, not $(fn\ f : t \Rightarrow f)\ 0$.

The types will be represented by this SML type:

```
datatype typ = VAR of string | INT | BOOL | ARROW of typ * typ
```

The abstract syntax trees (ASTs) representing this language will have the following SML type:

fun typeof env AST_ID s = env(s)
| typeof env AST_Num of int = INT
| typeof env AST_fun(

```
datatype term = AST_ID of string | AST_NUM of int
    | AST_BOOL of bool
    | AST_FUN of (string * typ * term)
    | AST_APP of (term * term)
    | AST_SUCC | AST_PRED | AST_ISZERO
    | AST_IF of (term * term * term)
```

As with most ASTs, this datatype does not represent parentheses as they appear in the source language; in other words, e and (e) have the same representation.

Environments

As it goes, the type checker must remember the types of the variables that have been declared. It does so by storing the types in a *type environment*, which is a mapping from names to types. This environment must be extendable to allow new variables to be bound, and it must be searchable, to allow the types of bound variables to be later retrieved. For example, consider this term:

```
    fn x : int => fn y : bool => x
```

We would start with an empty environment (). When we come to the outer function, we would add the relation (x,int) to our environment, and evaluate the body of the function in that context. Similarly, when we come to the inner function, we add (y,bool) to our environment, and evaluate its body in the further extended environment. Thus, when we come to the expression x, we check it in the environment ((x,int), (y,bool)). In order to determine the type of x, we merely look it up.

3 Typing

1. Write down ML expressions of type `typ` corresponding to the abstract syntax trees for each of the following type expressions:

    (a) `int`

(b) `int -> bool`

(c) `('a -> 'b) -> ('a -> 'b)`

2. Write down ML expressions of type term corresponding to the abstract syntax trees for each of the following expressions

   (a) `succ (pred 5)`

   (b) `if 7 then true else 5`

   (c) `fn a : int => f a a`

3. Typing is done with respect to an environment E, which is a mapping from identifiers to types; the environment gives the types of any free identifiers in the expression. Think of these as the variables that are defined somewhere higher up in the program.

   We express this using the following notation

   ```
   E |- e : t
   ```

   which can be read "from environment E, it follows that expression e has type t".

   Next we describe the actual typing rules.

   (a) A variable has the type the environment has stored for it.

   ```
                  E(x) = t
   (ID)          ----------
                 E |- x : t
   ```

   (b) An integer literal has type int and a Boolean literal has type bool.

   ```
    (NUM)         E |- n : int

   (TRUE)        E |- true : bool

   (FALSE)       E |- false : bool
   ```

   (c) The built-in `succ` and `pred` functions have type $int \to int$ and the built-in `iszero` function has type $int \to bool$.

```
 (SUCC)       E |- succ : int  ->  int

(PRED)       E |- pred : int -> int

(ISZERO)     E |- iszero :  int -> bool
```

(d) In an if-then-else, the condition must have type bool and the branches have to have the same type (but this can be any type).

```
            E |- e1 : bool      E |- e2 : t       E |- e3 : t
(IF)    ---------------------------------------------------
                   E |- if e1 then e2 else e3 : t
```

(e) For a function `fn x :  t1 => e`, if the body `e` has the type `t2` when we extend the environment by mapping `x` to `t1`, then the function has type $t1 \to t2$.

```
                  E[x : t1] |- e : t2
(-> INTRO) --------------------------------
             E |- fn x : t1 => e : t1 -> t2
```

(f) The ($\to$ INTRO) rule uses the notation E[x : t1] to denote an *updated environment* that is the same as E except that it maps x to t1.

When we apply a function e1 to an argument e2, the function must have type $t1 \to t2$ for some t1 and t2; then the argument must have type t1 and the application as a whole has type t2.

```
              E |- e1 : t1 -> t2       E |- e2 : t1
(-> ELIM) ---------------------------------------
                      E |- e1 e2 : t2
```

Given these rules, we can write a function $typeOf\ : env \to term\ \to typ$ which takes an environment, returns a function taking a term, and returns its type (if the term is well typed) or an error (if it isn't). A skeleton of the implementation is in *typechecker.sml*. It uses a type datatype and an environment implementation defined in *type.sml* .

Page #2

```
data type = INT | Bool | Arrow Type to turp

Type of : env → term → type

    fun type of env AST_ID s = env(s)
      | type of env AST_NUM of int = INT
      | type of env AST_Fun(string, typ, body) = Arrow(typ, type of (body))
      | type of env AST_If(e1, e2, e3) = if typeof (e1) == Bool
                                             t2 = typeof(env(e2))
                                             t3 = typeof(env(e3))
                                             if t2 == t3
                                                 t2
                                         else
                                             type error

    | typeof   env   AST_APP(e2 x e2) = let t2 = typeof(env, e1)
                                         in
                                         Case t1 of ARROW(input, output)
```

```
if typeof (env, $c_2$) == input
  then
     Output
  else
     type error
```