Data Structures and Functional Programming	**Problem Set 4**
CS 3110, Spring 2014	**Thursday, March 20**
Version: 1	**Last Modified: March 13, 2014**

Overview

In this assignment you will implement several functions over infinite streams of data, an algorithm for inferring types, and an interpreter for CS3110Caml, a simple functional language.

Objectives

- Develop representations of infinite streams and gain experience with lazy evaluation.
- Implement an interpreter for a simple functional language.
- Build a simple type inference engineering using unification.
- Gain familiarity with implementing pattern matching and exhaustiveness checking.

Recommended reading

The following supplementary materials may be helpful in completing this assignment:

- Lecture 13
- Real World OCaml, Chapters 1-10

What to turn in

You should submit your solutions in files streams.ml, eval.ml, infer.ml, and unify.ml. Any comments you wish to make can go in comments.txt or comments.pdf. If you choose to submit any Karma work, you may submit the file karma.ml (be sure to describe what you've done in the comments file).

Lazy Evaluation

A **stream** is an infinite sequence of values. We can model streams in OCaml using the type

```
type 'a stream = Stream of 'a * (unit -> 'a stream)
```

Intuitively, the value of type `'a` represents the current element of the stream and the `unit -> 'a stream` is a function that generates the rest of the stream.

"But wait!" you might ask, "why not just use a list?" It's a good question. We can create arbitrarily long finite lists in OCaml and we can even make simple infinite lists. For example:

```
# let rec sevens = 7 :: sevens;;
val sevens : int list =
  [7; 7; 7; 7; 7; 7; ...]
```

However, there are some issues in using these lists. Long finite lists must be explicitly represented in memory, even if we only care about a small piece of them. Also most standard functions will loop forever on infinite lists:

```
# let eights = List.rev_map ((+) 1) sevens;;
(loops forever)
```

In contrast, streams provide a compact, and space-efficient way to represent conceptually infinite sequences of values. The magic happens in the tail of the stream, which is evaluated lazily—that is, only when we explicitly ask for it. Assuming we have a function `map_str : ('a -> 'b) -> 'a stream -> 'b stream` that works analogously to `List.map` and a stream `sevens_str`, we can call

```
# let eights_str = map_str ((+) 1) sevens_str;;
val eights_str : int stream
```

and obtain an immediate result. This result is the stream obtained after applying the map function to the first element of `sevens_str`. Map is applied to the second element only when we ask for the tail of this stream.

In general, a function of type `(unit -> 'a)` is called a **thunk**. Conceptually, thunks are expressions whose evaluation has been delayed. We can explicitly wrap an arbitrary expression (of type `'a`) inside of a thunk to delay its evaluation. To force evaluation, we apply the thunk to a unit value. To see how this works, consider the expression

```
let v = failwith "yolo";;
Exception: Failure "yolo"
```

which immediately raises an exception when evaluated. However, if we write

```
let lazy_v = fun () -> failwith "yolo";;
val v : unit -> 'a = <fun>
```

then no exception is raised. Instead, when the thunk is forced, we see the error

```
v ();;
Exception: Failure "yolo"
```

In this way, wrapping an expression in a thunk gives rise to **lazy evaluation**.

Manipulating Streams

To specify a stream, we need to be able to determine how to generate its elements. One way to construct a stream is to use a function of the type

```
unfold : ('a -> 'b * 'a) -> 'a -> 'b stream
```

In contrast to `List.fold_left` and `List.fold_right`, which take a list and produce a value, the `unfold` function takes a generator function and a seed. The generator function tells us how to generate the next stream element and the next seed. As an example, we can create the stream of all natural numbers using

```
let nats = unfold (fun x -> x, succ x) 0
```

It is often easier to use two functions:

1. a function `gen_head : 'a -> 'b` that tells us how to construct a new element of the stream from a seed, and
2. a function `gen_seed : 'a -> 'a` that tells us how to construct the next seed.

This idea is captured in the function

```
univ : ('a -> 'b) * ('a -> 'a) -> 'a -> 'b stream
```

We can use this function to construct the natural numbers via

```
let nats = univ ((fun x -> x), succ) 0
```

Either way, we obtain a data structure that represents **all** of the natural numbers in a compact way.

Exercise 1:

Implement the `unfold` function as described above. More details can be found in the documentation.

Implementing Stream Functions

In each of the following exercises you may **not** use the `rec` keyword. Instead, you should use the `univ` function to create and manipulate streams.

Exercise 2:

- Implement a function

  ```
  repeat : 'a -> 'a stream
  ```

  that produces a stream whose elements are all equal to `x`.

- Implement a function

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

that (lazily) applies the input function to each element of the stream.

- Implement a function

```
diag : 'a stream stream -> 'a stream
```

that takes a stream of streams of the form

$$\begin{matrix} s_{0,0} & s_{0,1} & s_{0,2} & \cdots \\ s_{1,0} & s_{1,1} & s_{1,2} & \cdots \\ s_{2,0} & s_{2,1} & s_{2,2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

and returns the stream containing just the diagonal elements

$$s_{0,0}\, s_{1,1}\, s_{2,2} \cdots$$

- Write a suffixes function

```
suffixes : 'a stream -> 'a stream stream
```

that takes a stream of the form

$$s_0 \quad s_1 \quad s_2 \quad s_3 \quad s_4 \quad \cdots$$

and returns a stream of streams containing all suffixes of the input

$$\begin{matrix} s_0 & s_1 & s_2 & \cdots \\ s_1 & s_2 & s_3 & \cdots \\ s_2 & s_3 & s_4 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

- Implement a function

```
interleave : 'a stream -> 'a stream -> 'a stream
```

that takes two streams $s_0 s_1 \cdots$ and $t_0 t_1 \cdots$ as input and returns the stream

$$s_0\, t_0\, s_1\, t_1\, s_2\, t_2 \cdots$$

Creating Infinite Streams

Exercise 3:

In this exercise you will create some interesting infinite streams.

- The Fibonacci numbers a_i can be specified by the recurrence relation $a_0 = 0$, $a_1 = 1$ and $a_n = a_{n-1} + a_{n-2}$. Create a stream `fibs : int stream` whose nth element is the nth Fibonacci number.

- The irrational number π can be approximated via the formula

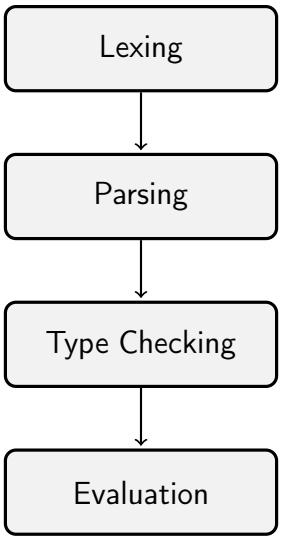
$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}.$$

  Write a stream `pi : float stream` whose nth element is the nth partial sum in the formula above.

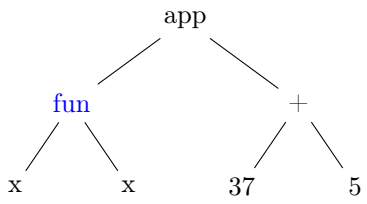
- The **look-and-say sequence** a_i is defined recursively – $a_0 = 1$ and a_n is generated from a_{n-1} by reading off the digits of a_{n-1} as follows: for each consecutive sequence of identical digits, pronounce the number of consecutive digits and the digit itself. For example:

  - the first element is the number 1,
  - the second element is 11 because the previous element is read as "one one",
  - the third element is 21 because 11 is read as "two one",
  - the fourth element is 1211 because 21 is read as "one two, one one",
  - the fifth element is 111221 because 1211 is read as "one one, one two, two one".

  Write a stream `look_and_say : int list stream` whose nth element is a list containing the digits of a_n ordered from most significant to least significant.



(a)



(b)

Figure 1: (a) Interpreter stages; (b) Abstract syntax tree for a simple expression

OCaml Interpreter

In this part of this assignment, you will implement an interpreter for a subset of OCaml called 3110Caml. This will provide functionality similar to the top-level loop we have been using for much of the semester.

Overview

An interpreter is a program that takes a source program and computes the result described by the program. An interpreter differs from a **compiler** in that it carries out the computations itself rather than emitting code which implements the specified computation.

A typical interpreter is implemented in several **stages** (see Figure 1 (a)):

1. Lexing: the interpreter splits the input program into **tokens**.
2. Parsing: the interpreter takes the list of tokens and converts it into an **abstract syntax tree** (AST) according to the grammar of the language.
3. Type checking: the interpreter verifies that the program is well-formed according to the type system for the language—e.g. the program does not contain expressions such as `3 + true`.
4. Evaluation: the interpreter evaluates the AST into a final **value** (or diverges).

As an example given the 3110Caml program `(fun x -> x) (37+5)`, the lexer would produce a stream of tokens:

The parser would produce the syntax tree shown in Figure 1 (b); the type checker would succeed, and evaluation would proceed as follows:

$$(\text{fun x -> x})\ (37{+}5) \rightarrow (\text{fun x -> x})\ 42 \rightarrow 42$$

We have provided you with a type Ast.expr that represents abstract syntax trees, as well as a lexer and parser that transforms into strings into ASTs[1]. Your task will be to implement type checking and evaluation.

Interacting with the Interpreter

Our 3110Caml interpreter is similar to the OCaml toplevel. To run the interpreter, type[2]

```
> cs3110 compile main.ml
> cs3110 run main.ml
```

You should see a list of commands followed by the prompt

```
zardoz #
```

You can then type expressions in to be evaluated by your interpreter. Unlike the OCaml toplevel, the 3110Caml interpreter only accepts one line expressions and you do not have to terminate expressions with the ;; . Here is a sample toplevel session:

```
zardoz # 1729
- : int = 1729
zardoz # fun x -> x
- : 'a -> 'a = <fun>
zardoz # let f = fun x -> x+1 in f 3109
- : int = 3110
zardoz # match [17;1729;5040] with [] -> false | x::xs -> true
- : bool = true
```

Our Subset of OCaml

The interpreter you will build will support a small but expressive subset of OCaml. 3110Caml has been carefully designed to help you understand the fundamentals of building an interpreter for a functional language without getting bogged down in some of the more complicated features of OCaml.

[1] If you are interested in understanding how these work, you may consult the Parser directory and check out the ocamlyacc parser generator.

[2] To improve your 3110Caml toplevel experience you may wish to install the rlwrap package and instead run rlwrap cs3110 run main.ml. This will enable you to use the arrow keys within the interpreter.

Values

3110Caml has five basic types of values:

1. Integers, which correspond to the OCaml type `int`;
2. Booleans, which correspond to the OCaml type `bool`;
3. Closures, which represent a functions and the variables in its scope;
4. Unit;
5. Lists.

Note that the interpreter does not support strings, tuples, or user-defined data types.

In addition, we will use a sixth internal value `VUndef` for creating recursive definitions. This value is not recognized by the parser and so is available to the programmer. You should ensure that your interpreter never evaluates an expression to `VUndef` (by raising an exception). When an expression is evaluated successfully, the result should have type `Ast.value`. Consult the documentation for more information.

Expressions

3110Caml supports a rich collection of expressions including `let` bindings, anonymous functions, conditionals, and `match` statements. The following code, taken from `ast.ml`, defines the `expr` type.

```
type expr =
    Constant of constant
  | BinaryOp of binary_op * expr * expr
  | UnaryOp of unary_op * expr
  | Var of id
  | Fun of id * expr
  | Cons of expr * expr
  | IfThenElse of expr * expr * expr
  | Let of id * expr * expr
  | LetRec of id * expr * expr
  | App of expr * expr
  | Match of expr * (pattern * expr) list
```

Note that 3110Caml supports pattern matching. However, because it does not support user-defined types, pattern matching here is mostly useful for deconstructing lists. Consequently, a `pattern` is either a constant, a variable, or a cons of two patterns:

```
type pattern =
  | PConstant of constant
  | PVar      of id
  | PCons     of pattern * pattern
```

If a given match expression does not match any of the patterns in a match expression, the pattern matching is said to be **inexhaustive**. Your implementation is not responsible for statically detecting inexhaustive match cases and should therefore throw an exception if pattern matching fails at run-time.

Roadmap

To complete this part of the assignment, you will have to implement the three core processes outlined above: evaluation, type annotation, and unification as outlined in the following exercises.

Exercise 4:

In this exercise you will implement the evaluation step of the interpreter, which is taken care of in the file `eval.ml`. In particular, you are responsible for the following:

a) Implement the function

```
eval : Ast.expr -> Ast.environment -> Ast.value
```

which evaluates 3110Caml expressions as prescribed by the **environment model**. More details on precisely how this is implemented can be found in these notes.

b) Implement the function

```
pattern_match : Ast.value -> Ast.pattern -> bool * Ast.environment
```

which determines dynamically (i.e. at runtime) whether or not a given value matches a particular pattern. This function should match your intuitive notion of how pattern matching works, but more precise information can be found in the documentation.

Exercise 5:

This exercise you will cover the type annotation phase of the interpreter, the first step in type inference. You are responsible for

a) Implementing the function

```
val annotate : Ast.expr -> (Ast.id, Ast.typ) Hashtbl.t -> Ast.aexpr
```

which annotates a given expression with proper type variables. The second argument is the **type context**, i.e. a mapping from the type variables to their associated types. In order to complete type annotation, you will also need to fill in the partially implemented function function `annotate_op`, a helper function that annotates the unary and binary operators.

The details of type annotation are explained in further detail later in this document as well as in the the documentation

b) In addition, implement the function

```
collect : Ast.aexpr list -> Ast.constr list -> Ast.constr list
```

which generates a list of constraints for unification, given an input expression. As with annotate, you will also need to complete the partially implemented function collect_operator_constraints, which collects the constraints from the unary and binary operators

The details of how these constraints are generated is outlined below, as well as in the documentation.

You will find the functions for this exercise in the file infer.ml.

Exercise 6:

In this exercise, you will complete type inference by implementing the function

```
unify : Ast.constr list -> Ast.substitution
```

in unify.ml. This function solves for a satisfying assignment of the type variables in the input constraint list. Further details on the unification algorithm can be found below.

Moving Forward

The remainder of this document outlines each of the above processes in greater detail to aid you in your implementation. Be sure to also consult the documentation for helper functions and modules that you may find useful in your implementation.

Implementing Type Inference

Type inference refers to the process of assigning types to expressions. Your interpreter will implement type inference in two stages:

1. Type annotation
2. Unification

Type annotation takes an expression as input and constructs an explicitly typed expression by analyzing the structure of the term. Unification takes two types as input and constructs a substitution that, when applied, makes the types equal, or fails if no substitution exists.

Type Annotation

Consider the OCaml program

```
(fun x y z -> if x (y * y) then z else x y)
```

Intuitively, we can infer the types of the expressions as follows:

- Because y is used in multiplication it must be of type int.
- Moreover, because x is applied to the argument (y * y) it must be some function. Then, because the output of the application is used in the conditional clause to an if statement, it must be of type bool. Consequently, x must have type int -> bool.
- Finally, z is used as the output of the if expression and therefore must have the same type as the output of the else expression, which is the output of x, i.e. bool.

Putting all these pieces together, the annotated expression would be

```
fun (x : int -> bool) (y : int) (z : bool) -> bool
```

The goal of type annotation is to formalize the above process and produce a set of equations between types that, if satisfiable, can be used to determine a type for the expression. We will call such annotations **constraints**.

During type annotation function, we will need to work with expressions containing type variables. Hence, we will introduce a context Γ to keep track of the current assignments of type variables to other types. Such a context can be represented concretely by a Hashtbl mapping identifiers to types. Hence, the overall type of annotate is as follows:

```
annotate : Ast.expr -> (Ast.id, Ast.typ) Hashtbl.t -> Ast.aexpr
```

The Ast.expr is the program to be annotated and the Hashtbl.t is the type context Γ. A detailed view of the rules for calculating constraints can be found in the documentation.

Unification

After annotation, we have an annotated expression, as well as a set of constraints between types that must be satisfied for the expression to be well-typed. To solve these constraints, we will use unification.

Consider the equation 'a -> ('a -> int) = int -> 'b. To unify this equation, we must find valid types to replace 'a and 'b with. The substitution that maps [('a$\mapsto$ int); ('b$\mapsto$int->int)] does this—we say this mapping **unifies** the constraint equation. Your task in this problem set is to implement the unification algorithm that generates such a mapping.

The solution, or mapping, that unifies the constraint equations is called a **type substitution**. (We really care about the case of substituting type variables with types, but note that type variables 'a and 'b are types just as well as int.) Formally, a function σ is a type substitution if it

satisfies the following properties:

$$\sigma(c) = c \text{ if } c \text{ is a constant type,}$$
$$\sigma(x) = \begin{cases} \tau & \text{if } x \mapsto \tau \in \sigma \\ x & \text{if } x \notin \text{dom}(\sigma) \end{cases},$$
$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau'),$$
$$\sigma(\tau\ list) = \sigma(\tau)\ list.$$

The algorithm to find such a substitution is given in the following diagram. Translate it to OCaml and you will have completed this section of the assignment.

Algorithm 1: Unification

input : A set C of constraints of the form $\tau = \tau'$.
output: A substitution, σ satisfying C, if one exists and an error otherwise
if $C = \emptyset$ **then**
  return []
else
  $(\tau = \tau') \cup C' \leftarrow C$
  case $\tau = \tau'$
    return unify(C')
  case $\tau = x$ **and** x **is not a free variable of** τ'**.**
    return unify$(C'\{\tau'/x\}) \circ [x \mapsto \tau']$
  case $\tau' = x$ **and** x **is not a free variable of** τ**.**
    return unify$(C'\{\tau/x\}) \circ [x \mapsto \tau]$
  case $\tau = \tau_0 \to \tau_1$ **and** $\tau' = \tau'_0 \to \tau'_1$
    return unify$(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\})$
  case $\tau = \tau_0$ list **and** $\tau' = \tau'_0\ list$
    return unify$(C' \cup \{\tau_0 = \tau'_0\})$
  case otherwise
    return An error indicating that the unification failed.

A word on notation: the composition operator $\circ$ is used to combine two substitutions, $C\{\tau/x\}$ means "replace all unbound occurences of x with τ within C", $[x \mapsto \tau]$ represents the function (that is, type substitution) σ that maps variable x to type τ, and $\cup$ is used to combine two sets of constraint equations.

Example of Type inference

Consider the expression from the type annotation section above:

```
(fun x y z -> if x (y * y) then z else x y)
```

We start with an pure expr AST and convert it to an annotated one, as shown below:

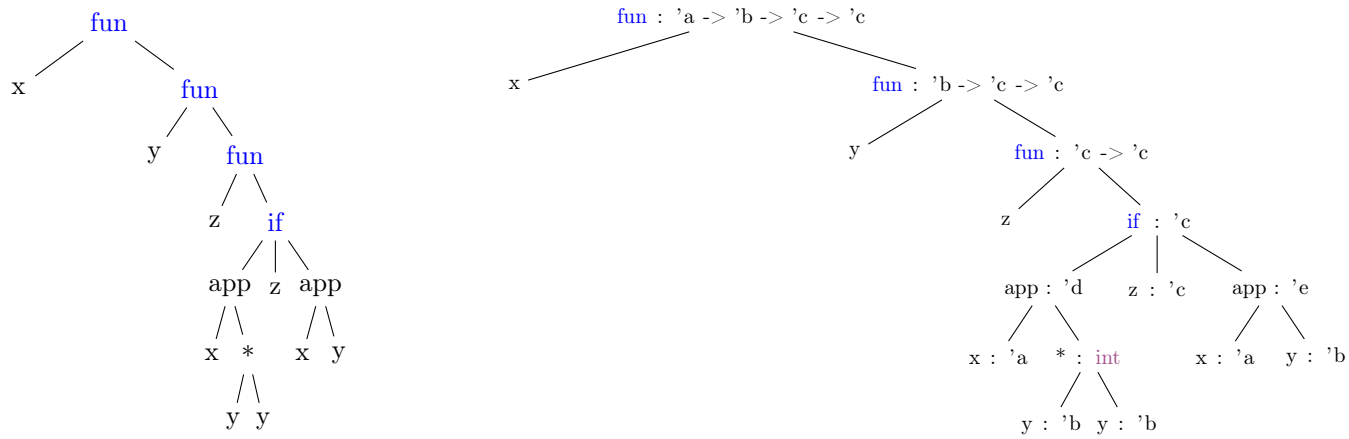


Figure 2: The annotation phase of type inference.

The above annotated AST should generate the following set of constraints:

$$\{\ \text{'a} = \text{'b -> 'c};\ \ \text{'b} = \text{int};\ \ \text{'a} = \text{int -> 'd};\ \ \text{'c} = \text{'e};\ \ \text{'d} = \text{bool}\ \}$$

We would then apply unification to the above constraints to generate the substitution

$$\sigma = \{\ (\text{'a} \mapsto \text{int -> bool});\ \ (\text{'b} \mapsto \text{int});\ \ (\text{'c} \mapsto \text{'e});\ \ (\text{'d} \mapsto \text{bool})\ (\text{'e} \mapsto \text{bool})\ \}$$

We then apply the substitution to get the final annotated AST:

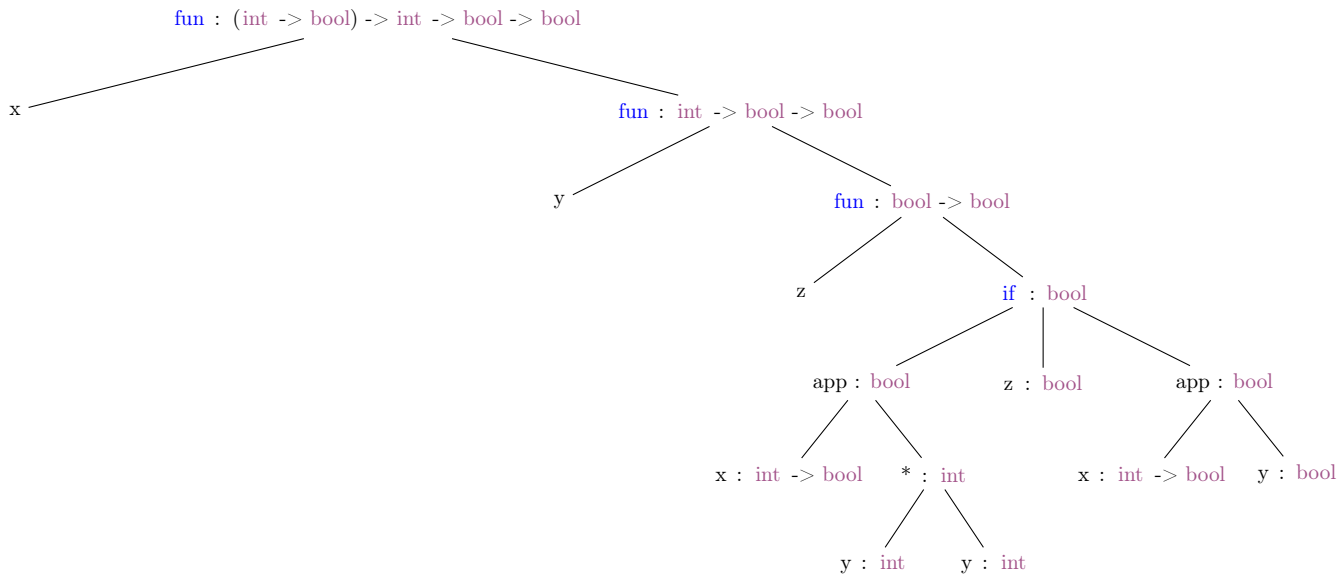


Figure 3: Final annotated AST after unification.

Unification may seem confusing at first, but we encourage you not to overthink it. You know how to solve these problems! This algorithm is similar to ones you have been solving all semester. It is just the formal, algorithmic process that OCaml uses to check for type errors.

Comments

We would like to know how this assignment went for you. Were there any parts that you didn't finish or wish you had done in a better way? Which parts were particularly fun or interesting? Did you do any Karma problems?

Karma suggestions

- Implement exhaustiveness checking for pattern matching.
- Extend your interpreter with pairs, or other data types.