Final Project
COMP 302 Programming Languages and Paradigms

Due Date: 30 April 2021

1 Introduction

In this project, you will implement a language called MiniML, in OCaml. This will allow you to work with a larger code base, make use of many of the concepts that you have seen in class throughout the semester, and gain a deeper understanding on key concepts such as free variables, substitution, evaluation, type checking and type inference. MiniML is similar to OCaml itself and very similar to the language that is discussed in the lectures and lecture notes; it is however also more powerful than the minimal language discussed in the lecture notes. In particular, it includes n-ary tuples (pairs written as $(e_1, \dots,\ e_n)$), a more general let-expression, functions, function application, and recursion.

Goal of the project

In this project, you will extend existing implementation with simple code analysis tools, an interpreter (which evaluates an expression), a type checker, and eventually, a type inference engine.

Overview

The general pipeline we will use in this project is as follows:

1. **Frontend** (From *External Language* to *Internal Language*): In the frontend, we read and parse a program written in an *external* language and produce the corresponding *internal* program (i.e. the program in the internal language). We already have implemented this frontend, which means you can conveniently write programs in the *external language*.
2. **Core** : The core of this project consists of code analyzer, an evaluator, a type checker, and a type inference engine. All these programs take a program in the *internal language* as an input. Hence, you will be writing programs that analyze and evaluate expressions in the *internal language*. The frontend translates MiniML programs into their corresponding internal representation and thus allows you to write MiniML programs in a clean, readable form.

We will explain this pipeline in more detail in the next section.

2 Internal and External Language

Our MiniML internal language can be described as an extension of the language that is given in the lecture notes:

$$
\begin{aligned}
\text{types } \tau &::= \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \texttt{ -> } \tau_2 \mid \tau_1 * \ldots * \tau_n \\
\text{expressions } e &::= \ldots \mid \mathsf{fn}\ x : \tau \texttt{ => } e \mid e_1\ e_2 \mid \mathsf{rec}\ f : \tau \texttt{ => } e \mid (e_1,\ \ldots,\ e_2) \mid \mathsf{let}\ ds\ \mathsf{in}\ e\ \mathsf{end} \\
\text{declaration } d &::= \mathsf{val}\ x = e \mid \mathsf{val}\ (x_1, \ldots, x_n) = e \mid \mathsf{name}\ x = e \\
\text{declarations } ds &::= \cdot \mid d\ ds
\end{aligned}
$$

It is worth noting that this grammar describes an *internal* language where issues regarding precedence order, brackets, etc. have been already resolved. A programmer actually writes programs in the *external* language, i.e. the language that the parser accepts. In the project, we provide a frontend which reads and parses programs written in the *external* language and produces the corresponding *internal* program (i.e. the program in the internal language). For example, we write

```
let fun fact (x : int) : int =
  if x = 0 then
    1
  else
    x * fact(x - 1)
in
  fact 5
end;
```

which corresponds to

$$\mathsf{let\ val\ fact} = \mathsf{rec\ fact} : \mathsf{int} \texttt{ -> } \mathsf{int} \texttt{ => } \mathsf{fn}\ x : \mathsf{int} \texttt{ => } \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x * fact(x - 1)\ \mathsf{in\ fact}\ 5\ \mathsf{end}$$

Having such a frontend makes it easier to write programs as input.

In general, a valid MiniML program is just an expression followed by a semicolon. All MiniML programs eventually compute to values (if they terminate). Our external language supports various operators. They include arithmetic operators like addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and negation (`~`), arithmetic comparisons (`<`, `>`, `<=`, `>=`), equality comparisons (`=`, `!=`) and logical operators like logic and (`&&`) and logic or (`||`). All operators have familiar precedence levels. Notice that equality comparison operators only apply to integers.

3 With A Little Help From Your Friendly Teaching Assistants . . .

3.1 Parsing

We provide a number of helper functions in order to help you focus on the essential part of the project.

Following is the signature of a parsing module which we provide:

```
module P : sig
  val parse : string -> (string, exp) either
end
```

You can invoke this function by

```
P.parse "some string"
```

The function returns `Right e` if the string is a valid MiniML program, or `Left s` if it is not and `s` is the corresponding (not-so-specific) error message. For example:

```
P.parse "let fun fact (x : int) : int =
  if x = 0 then
    1
  else
    x * fact(x - 1)
in
  fact 5
end;";;
(* this program returns
- Right (Let
 ([Val
    (Rec
      ("fact", TArrow (TInt, TInt),
       Fn
        ("x", Some TInt,
         If
          (Primop (Equals,
            [Var "x"; Int 0]),
          Int 1,
          Primop (Times,
           [Var "x";
            Apply (Var "fact",
             Primop (Minus,
              [Var "x"; Int 1]))])))),
    "fact")],
  Apply (Var "fact", Int 5)))
*)
```

In a negative case:

```
P.parse "1" (* should end with semicolon *) ;;
- : Left "Expected SEMICOLON token"
```

3.2 Pretty Printing

Clearly, abstract syntax is not very easy to read. Sometimes you want to quickly verify if an abstract syntax tree is as expected. In that case, you can use a printing helper which is provided by the following module:

```
module Print : sig
  val exp_to_string : exp -> string
  val typ_to_string : typ -> string
end
```

which converts an `exp`/`typ` to a readable form:

```
let Right e = P.parse "let fun fact (x : int) : int =
  if x = 0 then
    1
  else
    x * fact(x - 1)
in
  fact 5
```

```
end;" in
print_string Print.exp_to_string e;
print_newline ();;
(* this program prints

let fun fact : int -> int x = if x = 0 then 1 else x * fact (x - 1) in fact 5 end

*)
```

3.3 Testing

We have two methods to help you test your code, unit tests and hidden tests.

We implement some basic helper functions to unit test your code. For a requested function `foo`, you can write your own unit tests in `foo_tests` and run your tests by evaluating your code and executing `run_foo_tests ()`. For example, for Q1, you can write some unit tests in `free_vars_tests` and run `run_free_var_tests ()` to run all the tests. You are encouraged to test your code frequently!

Hidden tests can be run by clicking the "Grade" button. The report will simply tell you either that you pass all the tests or the number of tests you fail. The tests are here to catch some common errors and do not meant to be exhaustive. You should write more unit tests to make sure your code is correct!

At last, please remember to **back up your code!** You will be working on this project for days and weeks. You don't want to restart everything from scratch because LearnOCaml erases your work!

3.4 Other Helpers

We provide other helper definitions in the prelude buffer above the main buffer. It includes some useful helper functions as well as some examples of valid MiniML programs. Make sure to take a quick look before getting started!

4 Grading

We provide the testing mechanisms in order to help you focus on the technical problems themselves. **This project is graded offline.** That is, your code might pass all the hidden tests on LearnOCaml, but that does **not** imply a guaranteed 100%; bugs might still get caught by our offline tests and our manual code review. For this reason, please try your best to **test your own code!**

You **can** organize the given code freely, including defining your own top-level definitions, adding or removing `rec`, etc, as long as you implement the target functions correctly with the exact given types.

5 Questions

5.1 Q0 (0 point): Trying out Parser Frontend

Get yourself familiar with the external syntax of MiniML and the usage of the parsing function `P.parse`. Please use `parse_tests : (string * (string, exp) either) list` to familiarize yourself. Each entry is a MiniML program and the expected output of `P.parse`. Hit Grade button and see if you have got the tests right.

5.2 Q1 (10 points): Free Variables

Implement the function `free_vars : exp -> name list` which when given an expression computes the free variables occurring in the expression. This function is key to implementing substitution properly.

Your implementation should follow closely the definition of free variables given in the notes and extend it to tuples, let expressions, functions and recursion. The order of the returned names does not matter.

5.3 Q2 (15 points): Unused Variables

OCaml also tests whether a given variable is used or not; this is a useful little tool which we would also like to support for MinML. Implement a function `unused_vars: exp -> name list` which given an expression checks for each variable introduced by a binding construct (i.e. a function, let expression, etc.) whether the variable the construct introduces is in fact used in the body of the expression.

For example:

`let val x = 3 in 4 end;`	`x` is unused
`let val x = true in let val y = 4 in x + 5 end end;`	`y` is unused
`let val x = 3 in let val x = 4 in x + x end end;`	`x` (the first occurrence) is unused

Similarly, given the following program, `x` and `test` is unused:

```
let fun test (x : int) : int = 3 in 4 end;
```

Implement the function `unused_vars: exp -> name list`. It traverses an expression and for every construct which introduces some bound variables (i.e. functions, let-expression) we will check whether these variables occur free in the body; if they do, then indeed these variables are used; if they do not occur free in the body, then the variables are unused. You might want to use `free_vars` that you just implemented. The order of names does not matter.

5.4 Q3 (15 points): Substitution

Finish the implementation of function `subst:(exp * name) -> exp -> exp` for substitution to handle let-name, pairs, let-pair, functions, function application, and recursion.

The first argument to the function `subst` contains a tuple of an expression `e'` and a variable name `x` denoting the substitution `[e'/x]`. The second argument is an expression `e` to which we apply the substitution. In other words,

```
subst (e',x) e = [e'/x]e
```

i.e. the function `subst` will replace any free occurrence of the variable `x` in the expression `e` by the expression `e'`. Note that `subst` is **capture-avoiding**. Use `fresh_var` to obtain a fresh variable name if necessary.

You can then for example test your substitution function as follows:

```
# subst (Int 5, "x") (If (Bool(true), Var "x", Var "y"));
val it = If (Bool true, Int 5, Var "y") : exp
```

5.5 Q4 (30 points): Big Step Evaluation

In this question, we will implement an interpreter based on *big-step evaluation*. Your task is to implement the missing cases of the function `eval` for big-step evaluation. In particular, add the cases for functions, function application, recursion, and handling generalized let-expressions, i.e. those cases which currently raise the exception `NotImplemented`. The evaluation rules are fully listed in Figure 1.

If you encounter a situation not covered by these rules, then you should `raise (Stuck "message")` (with something more descriptive than `"message"`). See the other cases for examples of this.

The evaluator is **not** environment-based, so there is a direct translation from the above rules to code. Use `subst` that you just implemented.

As is often the case, not very much code is required.

5.6 Q5 (25 points): Type Checking

In this question, we are inferring the type of an expression. We use the type annotations on recursion and functions, to guarantee that we can always uniquely infer a unique type of an expression. It simplifies the type inference problem. For now, you can ignore type variables.

To describe the type inference we will use the following judgement:

$$\Gamma \vdash e \Rightarrow T \qquad \text{Infer type } T \text{ for expression } e \text{ in context } \Gamma$$

The inference rules are given in Figure 2. Your implementation of the function `infer: context -> exp -> typ` should follow exactly the algorithm described by the typing rules. It takes as input a context describing the typing assumptions and an expression. It returns the type of the expression. *For this question, you can assume that all functions are annotated with their types. We will handle those that are not in the next question.*

If an expression is not well-typed, then we can raise a `TypeError` exception by calling the `type_fail` function.

5.7 Q6 (30 points): Unification

Lastly, we want to enable full type inference; as a consequence, we can omit the type annotations on functions and recursion. Key to type inference is unification. Your task is to implement the function `unify: typ -> typ -> unit`, which checks whether two types are unifiable, i.e. if we can make them syntactically equal.

You should follow the description of unification in the class notes. Type variables are modeled via references. An uninstantiated type variable is modeled as a pointer to a cell with content `None`. In other words to create a new type variable we can simply use a function

```
let fresh_tvar () = TVar (ref None)
```

Once we know what a type variable should be instantiated with we simply assign it the correct type. For example, if we have a type variable `TVar x`, then `x` has type `(typ option) ref` and we can replace every occurrence of `TVar x` by the type `Int` using assignment `x := Some TInt`.

This will destructively update the type variable `x` and directly propagate the instantiation for it. No extra implementation of a substitution function is necessary to propagate instantiations.

Implement the function `unify : typ -> typ -> unit`. If two types are unifiable, they will be denoting the same type after unification succeeds. If unification fails, raise a `TypeError` exception by

$e \Downarrow v$ expression e evaluates to value v

$$\frac{e \Downarrow v}{\text{let } \cdot \text{ in } e \text{ end} \Downarrow v} \quad \text{(B-NO-DECS)}$$

$$\frac{e1 \Downarrow v_1 \qquad [v_1/x](\text{let decs in } e \text{ end}) \Downarrow v}{\text{let val } x_1 = e_1 \text{ decs in } e \text{ end} \Downarrow v} \quad \text{(B-LET-VAL)}$$

$$\frac{[e_1/x](\text{let decs in } e \text{ end}) \Downarrow v}{\text{let name } x_1 = e_1 \text{ decs in } e \text{ end} \Downarrow v} \quad \text{(B-LETN)}$$

$$\frac{e1 \Downarrow (v_1, \ldots, v_n) \qquad [v_1/x_1, \ldots, v_n/x_n](\text{let decs in } e \text{ end}) \Downarrow v}{\text{let val } (x_1, \ldots, x_n) = e_1 \text{ decs in } e \text{ end} \Downarrow v} \quad \text{(B-LET-TUPLE)}$$

$$\frac{\text{for all } i \;\; e_i \Downarrow v_i}{(e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_m)} \quad \text{(B-TUPLE)}$$

$$\frac{[\text{rec } f : \tau \text{ => } e/f]e \Downarrow v}{\text{rec } f : \tau \text{ => } e \Downarrow v} \quad \text{(B-REC)}$$

$$\frac{e_1 \Downarrow \text{fn } x \text{ => } e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v} \quad \text{(B-APP)}$$

$$\frac{}{\text{fn } x \text{ => } e \Downarrow \text{fn } x \text{ => } e} \quad \text{(B-FN)}$$

$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \quad \text{(B-IFTRUE)}$$

$$\frac{e_1 \Downarrow \text{false} \qquad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \quad \text{(B-IFFALSE)}$$

$$\frac{e \Downarrow v}{e : \tau \Downarrow v} \quad \text{(B-ANNO)}$$

$$\frac{n \text{ is a number}}{n \Downarrow n} \qquad \frac{b \in \{\text{true}, \text{false}\}}{b \Downarrow b}$$

$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v}{e_1 \text{ \&\& } e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \text{false}}{e_1 \text{ \&\& } e_2 \Downarrow \text{false}} \qquad \frac{e_1 \Downarrow \text{false} \qquad e_2 \Downarrow v}{e_1 \text{ || } e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \text{true}}{e_1 \text{ || } e_2 \Downarrow \text{true}}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \text{op is a binary operator}}{e_1 \text{ op } e_2 \Downarrow v_1 \text{ op } v_2} \text{ B-OP} \qquad \frac{e \Downarrow v}{\sim e \Downarrow -v}$$

Figure 1: rules for big-step evaluation

Inference Rules

$$\frac{\Gamma \vdash e \Rightarrow \mathsf{bool} \qquad \Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Rightarrow \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow \tau} \quad \text{(T-IF)}$$

$$\frac{\Gamma, x : \tau \vdash e \Rightarrow \tau'}{\Gamma \vdash (\mathsf{fn}\ x : \tau\ \texttt{=>}\ e) \Rightarrow \tau\ \texttt{->}\ \tau'} \quad \text{(T-FN)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n \Rightarrow \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) \Rightarrow (\tau_1 * \cdots * \tau_n)} \quad \text{(T-TUPLE)}$$

$$\frac{\Gamma, f : \tau \vdash e \Rightarrow \tau}{\Gamma \vdash (\mathsf{rec}\ f : \tau\ \texttt{=>}\ e) \Rightarrow \tau} \quad \text{(T-REC)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau\ \texttt{->}\ \tau' \qquad \Gamma \vdash e_2 \Rightarrow \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau'} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash decs \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \Rightarrow \tau}{\Gamma \vdash \mathsf{let}\ decs\ \mathsf{in}\ e\ \ \mathsf{end} \Rightarrow \tau} \quad \text{(T-LET)}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau} \quad \text{(T-ANNO)}$$

$$\frac{n \text{ is a number}}{\Gamma \vdash n \Rightarrow \mathsf{int}} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b \Rightarrow \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_i \Rightarrow \mathsf{int} \qquad \text{op is } \texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/}}{\Gamma \vdash e_1\ \text{op}\ e_2 \Rightarrow \mathsf{int}} \qquad \frac{\Gamma \vdash e_i \Rightarrow \mathsf{int} \qquad \text{op is } \texttt{<},\ \texttt{<=},\ \texttt{>=},\ \texttt{>=},\ \texttt{=},\ \texttt{!=}}{\Gamma \vdash e_1\ \text{op}\ e_2 \Rightarrow \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_i \Rightarrow \mathsf{bool} \qquad \text{op is } \texttt{\&\&},\ \texttt{||}}{\Gamma \vdash e_1\ \text{op}\ e_2 \Rightarrow \mathsf{bool}} \qquad \frac{\Gamma \vdash e \Rightarrow \mathsf{int}}{\Gamma \vdash \sim e \Rightarrow \mathsf{int}}$$

Rules for declarations

$$\frac{\Gamma \vdash dec_1 \Rightarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash decs \Rightarrow \Gamma_2}{\Gamma \vdash dec_1\ decs \Rightarrow \Gamma_1, \Gamma_2} \quad \text{(T-DECS)}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\mathsf{val}\ x\ \texttt{=}\ e) \Rightarrow (x : \tau)} \quad \text{(T-BY-VAL)}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\mathsf{name}\ x\ \texttt{=}\ e) \Rightarrow (x : \tau)} \quad \text{(T-BY-NAME)}$$

$$\frac{\Gamma \vdash e \Rightarrow (\tau_1 * \cdots * \tau_n)}{\Gamma \vdash (\mathsf{val}\ (x_1, \ldots, x_n)\ \texttt{=}\ e) \Rightarrow (x_1 : \tau_1), \ldots, (x_n : \tau_n)} \quad \text{(T-BY-VAL-TUPLE)}$$

Figure 2: Type inference rules

calling the `type_fail` function. Follow the algorithm described in the notes to unify two types, and fill in the implementation for `unify`.

5.8 Q7 (10 points) EXTRA CREDIT: Type Inference

Modify your function `infer` that it supports type inference. This is quite easy: if the type annotation in recursion and functions is `None` and we don't know the type of a variable `x` we want to add to the context, we simply generate a fresh type variable `tv` and add to the context the fact that `x` has type `tv`; instead of checking for equality, you call unification.