Assignment 3: Type Checking and Inference

CS 442/642

Due February 27th in class

For this assignment, your programs will be written in Standard ML. Either download Standard ML of New Jersey (`http://www.smlnj.org`) or use the `sml` command on the student.cs environment. The major task of this assignment is to implement a type inferencer for Milner,[1] a language based on the polymorphic λ-calculus.

Part A—System F

Page 71 of the course notes has definitions for $[\![car]\!]$ and $[\![cdr]\!]$ for pairs in System F, followed by a line that says, "The reader should verify ..." Here is your chance to verify. Show that $[\![car]\!]$ has the type given in the notes by providing a type derivation for the function $[\![car]\!]$ using the type rules of System F. The format of your solution should resemble that of the last part of Assignment 2. Invent and use shorthand as necessary to clarify your presentation, but make sure that all of the necessary details are present.

Milner Syntax

Milner is based on the Hindley-Milner polymorphic λ-calculus, including `let`-polymorphism, and extended with an `if` construct for conditional computation and a `fix` construct for recursive expressions (`fix (`x`, `E`)` computes the least fixed-point of $\lambda x.E$). The language has simple integer and boolean types and a small set of primitive functions for manipulating these. The following abstract syntax describes Milner:

```
<expr> ::= <abs> | <fix> | <app> | <let> | <cond> | <var> | <val>
<abs>  ::= Abs(<any string>, <expr>)
<fix>  ::= Fix(<any string>, <expr>)
<app>  ::= App(<expr>, <expr>)
<let>  ::= Let(<any string>, <expr1>, <expr2>)
<cond> ::= If(<expr1>, <expr2>, <expr3>)
<var>  ::= Var(<any string>)
<val>  ::= Int(<any int>) | Bool(<any bool>) | Prim(<any prim>)
<prim> ::= add | neg | mult | div | and | or | not | eq | lt | gt
```

The file

`http://www.student.cs.uwaterloo.ca/~cs442/W24/types.sml`

contains some start-up code for use on this assignment. You are free to use and/or modify it as you see fit. Indeed, parts of it will need to be modified in order to complete the assignment.

[1] A fictitious language invented for this course.

Part B

Start by implementing a pretty-printer for types. Write a function called `pptype` that takes in a type and outputs it in a human-friendly format as a string. For example,

```
pptype (Arrow(Arrow(TVar "a", TVar "b"), Arrow(TInt, TIint)))
```

should produce the string

```
"(('a -> 'b) -> (int -> int))"
```

Part C

You will implement, in ML, a type inferencer for Milner expressions. In particular, you are to implement algorithm W to create a type inferencer that enforces the following type rules (note: τ ranges over monotypes and σ ranges over polytypes):

$$\frac{A(\mathtt{x}) = \tau}{A \vdash \mathtt{x} : \tau}\ \text{[var]} \qquad \frac{A \vdash \mathtt{E}_1 : \mathtt{bool} \quad A \vdash \mathtt{E}_2 : \tau \quad A \vdash \mathtt{E}_3 : \tau}{A \vdash \mathtt{If(E_1,\ E_2,\ E_3)} : \tau}\ \text{[if]}$$

$$\frac{\langle \mathtt{x}, \tau_1 \rangle + A \vdash \mathtt{E} : \tau_2}{A \vdash \mathtt{Abs(x,\ E)} : \tau_1 \to \tau_2}\ \text{[abs]} \qquad \frac{\langle \mathtt{x}, \tau \rangle + A \vdash \mathtt{E} : \tau}{A \vdash \mathtt{Fix(x,\ E)} : \tau}\ \text{[fix]}$$

$$\frac{A \vdash \mathtt{E}_1 : \tau_1 \to \tau_2 \quad A \vdash \mathtt{E}_2 : \tau_1}{A \vdash \mathtt{App(E_1,\ E_2)} : \tau_2}\ \text{[app]} \qquad \frac{A \vdash \mathtt{E}_1 : \sigma \quad \langle \mathtt{x}, \sigma \rangle + A \vdash \mathtt{E}_2 : \tau}{A \vdash \mathtt{Let(x,\ E_1,\ E_2)} : \tau}\ \text{[let]}$$

$$\frac{A \vdash \mathtt{E} : \forall \alpha . \sigma}{A \vdash \mathtt{E} : \sigma[\tau/\alpha]}\ \text{[spec]} \qquad \frac{}{A \vdash \mathtt{true}\ \mathit{or}\ \mathtt{false} : \text{bool}}\ \text{[bool]}$$

$$\frac{A \vdash \mathtt{E} : \sigma}{A \vdash \mathtt{E} : \forall \alpha . \sigma}\ (\alpha \text{ not free in } A)\ \text{[gen]} \qquad \frac{}{A \vdash n \in \text{integers} : \text{int}}\ \text{[int]}$$

Your initial set of type assumptions should include at least the following bindings for Milner's predefined primitive functions:

1. addition: `add:  int -> (int -> int)`
2. numeric negation: `neg:  int -> int`
3. multiplication: `mult:  int -> (int -> int)`
4. division: `div:  int -> (int -> int)`
5. conjunction: `and:  bool -> (bool -> bool)`
6. disjunction: `or:  bool -> (bool -> bool)`
7. logical negation: `not:  bool -> bool`
8. numeric equality: `eq:  int -> (int -> bool)`
9. less than: `lt:  int -> (int -> bool)`
10. greater than: `gt:  int -> (int -> bool)`

Define a variable `initenv` that contains your initial environment, so that the TA may use it. Also define a variable `emptyenv` that represents the empty environment, so that the TA may also use that environment during testing.

Your top-level (i.e. "main") function should be called `W`. The TA will run your code by executing `W` *expr env* (for example, `W (Abs(Var x, Var x)) initenv`). Be sure that the initial and empty environments outlined above are available to the TA.

As on the previous assignment, you may use assignment to generate unique variable names—code for doing this has been provided for you.

In addition to being able to find the correct type for well-typed expressions, your submission must be able to handle input that is syntactically valid but ill-typed. If an expression is ill-typed, raise an exception. Do **not** crash the program.

The Inference Algorithm W

Algorithm *W* is presented on pages 72–76 of your course notes, but is reproduced here for your convenience.

```
W(A, x) = <[], tau>                            where tau = lookup(x, A)

W(A, Abs(x, E)) =
  let
    <S, tau> = W((x,alpha)+A, E)      alpha is a new type variable
  in
    <S, (alpha S)->tau>

W(A, App(E1, E2)) =
  let
    <S1, tau1> = W(A, E1)
    <S2, tau2> = W(A S1, E2)
    S3         = unify(tau1 S2, tau2->alpha)      alpha is new
  in
    <S3 o S2 o S1, alpha S3>

W(A, Let(x, E1, E2)) =
  let
    <S1, tau1> = W(A, E1)
    <S2, tau2> = W(((x,tau1')+A) S1, E2)
    where tau1' = tau1 with all variables not in (A S1) quantified
  in
    <S2 o S1, tau2>

W(A, If(E1, E2, E3)) =
  let
    <S1, tau1> = W(A, E1)
    S2         = unify(bool, tau1)
    <S3, tau3> = W(A (S2 o S1), E2)
    <S4, tau4> = W(A (S3 o S2 o S1), E3)
    S5         = unify(tau3 S4, tau4)
  in
    <S5 o S4 o S3 o S2 o S1, tau4 S5>

W(A, Fix(x, E)) =
  let
    <S1, tau1> = W((x,alpha)+A, E)    alpha is a new type variable
    S2         = unify(alpha S1, tau1)
  in
    <S2 o S1, alpha (S2 o S1)>
```

Type assumptions use the following syntax and definitions.

```
(x,tau)+A = A extended with the name x bound to type tau

lookup (x, []) = error
lookup (x, (y,tau)+A) = lookup (x,A)
lookup (x, (x,tau)+A) = tau'
  where tau' = tau with all quantified variables replaced with new ones
```

Part D—Demonstration

Write a suite of Milner programs:

- for each distinct location in your type inference program where an error could be produced, write a Milner program that would trigger an error at that point;
- write a well-typed Milner program that introduces a `let`-bound entity that is used polymorphically—that is, it is instantiated to different types at different locations.

For all of these programs, submit a one-page (total) transcript of the output of your type inference program on them. For the well-typed program, submit on paper a type derivation for your program according to the rules of Milner.

Note: There is no need for these programs to be long. Aim for simplicity.

Part E—Extending the Type System

For this part of the assignment, you are to extend your type inference system to support exception handling. To do this, you will introduce the primitive type `exception` and the following primitives and syntax:

- `Letex(<var>, <expr>)`—evaluates `<expr>` with `<var>` bound to an exception
- `Raise(<expr>)`—raises the exception `<expr>`
- `Handle(<expr1>, <expr2>, <expr3>)`—returns the value of `<expr2>` unless the exception `<expr1>` is raised during evaluation of `<expr2>`, in which case it returns the value of `<expr3>`

Note that you do not have to actually *implement* exception handling; you only have to be able to perform *type inference* on expressions with exceptions. Here is a sample Milner program that includes exceptions:

```
Letex("divzero",
   Let ("safediv", Abs("x", Abs("y",
           If (App (App (Prim Eq, Var "y"), Int 0),
               Raise(Var "divzero"),
               App (App (Prim Div, Var "x"), Var "y")
           )
        )),
        Handle(Var "divzero",
               App( App (Prim Add, App(App (Var "safediv", Int 8), Int 4))
                                 App(App (Var "safediv", Int 3), Int 0)),
               Int 0)
       )
)
```

How you support these primitives is up to you. For some of them, you may choose to extend W to handle them directly; for others, you may prefer to assign them types and place them in the initial type environment. However you choose to do it, you should submit, on paper, a brief description of your approach, which should include, at least, the following information:

- for every primitive that was implemented by extending W, you should include its associated type rule and pseudocode for the extension, in the style presented on page 4 of this document;
- for any primitive that was implemented by placing it in the type environment, you should indicate its type.

Notes

1. Spend some time familiarizing yourself with the provided `datatype` declarations, and construct some expressions satisfactory to the ML type checker, before starting to program.
2. Program and test substitutions and unification before starting on W.
3. Remember that `let` introduces generalizable type variables which can be instantiated differently in the same expression.
4. Part E is not as hard as it might look. Start by trying to determine what the necessary type rules should be. Then consider how they might be implemented. It may help to use the rest of the assignment as an example.

Submission

Submit your solution to parts B, C, and E as a single program. Submit an electronic copy of your source via `submit`. Also, submit a paper copy of your source in class on the due date. Your paper submission should also include a type derivation and one-page transcript for part D, and a brief design document for your solution for part E (maximum one page). The assignment is due by the beginning of class on the stated due date.

The distribution of marks for this assignment is expected to be 20% for part A, 10% for part B, 30% for part C, 20% for part D, and 20% for part E.