Computer Science 131, Spring 2001
Assignment 5: Typechecking

Out: Thursday, October 25
Due: Friday, November 2, 5:00pm

Instructions

For this assignment you must again use a special version of the SML/NJ compiler; the command on turing is /cs/cs131/bin/sml-cs131-a5.

Your code should be put in a file named assign5.sml file and submitted via cs131submit as usual. As always, your file should contain no syntax or type errors.

Introduction

For this assignment, you are to implement a typechecker for a language which is very like a subset of Standard ML. The major differences are that the allowed grammar covers only a small but nontrivial subset of SML, there is no polymorphism, and there is no interesting type inference.

The syntax for allowed programs is shown in Figure 1; the sml-cs131-a5 binary again includes a function

\[ \texttt{M.parse : string -> A.exp} \]

which takes a filename and parses the contents of the file into an SML representation of the abstract syntax. The signature of the A structure, which includes the datatype representation and some useful helper functions, is shown in Figure 2.

This datatype should be mostly self-explanatory. You can also try running various pieces of source code through the parser to see what comes out. Alternatively, Figure 3 gives a formal definition of the translation between abstract syntax into the SML representation. The notation \([e]\) will be used to denote the SML representation corresponding to the abstract syntax for \(e\). Note that anonymous functions (fn) must have a type annotation on their argument variable, and that function declarations are annotated with the argument type and the return type. (Other annotations on variables declared by val or the return type of anonymous functions are not necessary; we can always determine these from the type of the definition or the function body respectively, which we’d have to compute anyway.)
The binary also contains a structure `Env` which implements environments; the signature of this structure is

```sml
sig
  type 'a env
  val empty : 'a env
  val insert : 'a env * string * 'a -> 'a env
  val lookup : 'a env * string -> 'a
end
```

**Problem Statement (100%)**

Write a function

```sml
typeof : (A.ty Env.env) * A.exp -> A.ty
```

which given a typing environment and an expression, returns the expression’s type if one exists and raises the `Typeof` exception otherwise. (The SML code for this is `raise Typeof`)

You should use an auxiliary function

```sml
typeof_decl : (A.ty Env.env) * A.decl -> A.ty Env.env
```

for typechecking declarations. This should take a typing environment and a declaration, and if the declaration typechecks should return the given environment extended with the type of the variable or function defined in that declaration. In a sequence of declarations, each declaration can then be typechecked in an environment that includes the types of all previously-defined variables in the sequence.

Your typechecker should obey the following:

- Arithmetic and comparison operations can be performed on two integers and two reals, but not on one of each.

- The expression `real e` expects `e` to be of type `int` and returns the equivalent value of type `real`. There should be no subtyping or implicit conversions of any sort.

- Function definitions using `fun` can be recursive. (This means that to typecheck the definition you have to `assume` the function has the given argument and result type, and `assume` the argument has the given type, and check that under these assumptions the result has the specified type.) Definitions using `val` may not refer to themselves, so do not extend the environment in this case before typechecking the right-hand-side.

You do *not* have to write an evaluator for this language!

**Error Messages (10% EXTRA CREDIT)**

In addition to raising an exception when an error is found, have your typechecker print a helpful error message explaining what the problem was.
Subtyping (50% EXTRA CREDIT)

Write a function

\[
\text{typeof} : (A.ty \ Env.env) \times A.exp \rightarrow A.ty
\]

which returns the most-precise type of the given expression assuming a subtyping relationship that starts with \( \text{int} \leq \text{real} \) and the standard rules for subtyping between pair types and between function types. A very good starting point is the typechecker without subtyping.

Although you may still do pattern-matching to check that certain expressions have pair types or have function types, the \text{typeof} function should \textit{never} compare two types for equality; checking that one type is a subtype of the other (via \text{A.subtype}) is always sufficient. The \text{A.lub} function is useful when typechecking conditional expressions.

---

\[ e ::= \begin{array}{l}
    n \hspace{1cm} \text{(integers)} \\
    r \hspace{1cm} \text{(real numbers)} \\
    f, x, \ldots \hspace{1cm} \text{(variables)} \\
    \text{true} | \text{false} \\
    \text{fn} \ x : t \Rightarrow e \\
    (e_1, e_2) \\
    e_1 + e_2 \\
    e_1 - e_2 \\
    e_1 \times e_2 \\
    e_1 \leq e_2 \\
    e_1 = e_2 \\
    \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
    \text{let } d_1 \ldots d_n \text{ in } e \\
    e_1 e_2 \\
    \#1 e \\
    \#2 e
\end{array} \]

\[ d ::= \begin{array}{l}
    \text{val } x : t = e \\
    \text{fun } f(x : t_1) : t_2 = e
\end{array} \]

\[ t ::= \begin{array}{l}
    \text{int} \\
    \text{bool} \\
    \text{real} \\
    t_1 \rightarrow t_2 \\
    t_1 \times t_2
\end{array} \]

---

Figure 1: ML Subset for this Assignment
sig
  type var = string
  datatype ty = Int_t
  | Real_t
  | Bool_t
  | Arrow_t of ty * ty
  | Times_t of ty * ty
  datatype aop = Plus | Minus | Times
  datatype cop = Less | Equal | LessEq
  datatype exp = Int of int
  | Real of real
  | Bool of bool
  | Var of var
  | ToReal of exp
  | Arith of exp * aop * exp
  | Compare of exp * cop * exp
  | If of exp * exp * exp
  | Let of decl list * exp
  | Fn of var * ty * exp
  | Apply of exp * exp
  | Pair of exp * exp
  | Fst of exp
  | Snd of exp
  and decl = Val_d of var * exp
  | Fun_d of var * var * ty * ty * exp

val toString : exp -> string
val ty toString : ty -> string
val decl toString : decl -> string
val decls toString : decl list -> string

val subtype : ty * ty -> bool (* based on Int_t \subseteq Real_t *)
val lub : ty * ty -> ty option (* least upper bound, if any *)
val glb : ty * ty -> ty option (* greatest lower bound, if any *)
end

Figure 2: Signature of the structure A
\[\begin{align*}
\text{int} &= A.\text{Int}_t \\
\text{bool} &= A.\text{Bool}_t \\
\text{real} &= A.\text{Real}_t \\
[t_1 \to t_2] &= A.\text{Arrow}_t([t_1],[t_2]) \\
[t_1 \ast t_2] &= A.\text{Times}_t([t_1],[t_2]) \\
[n] &= A.\text{Int}(n) \\
[r] &= A.\text{Real}(r) \\
\text{true} &= A.\text{Bool}(true) \\
\text{false} &= A.\text{Bool}(false) \\
[x] &= A.\text{Var}("x") \\
\text{real } e &= A.\text{ToReal}([e]) \\
[e_1 + e_2] &= A.\text{Arith}([e_1],A.\text{Plus},[e_2]) \\
[e_1 - e_2] &= A.\text{Arith}([e_1],A.\text{Minus},[e_2]) \\
[e_1 \ast e_2] &= A.\text{Arith}([e_1],A.\text{Times},[e_2]) \\
[e_1 < e_2] &= A.\text{Compare}([e_1],A.\text{Less},[e_2]) \\
[e_1 \leq e_2] &= A.\text{Compare}([e_1],A.\text{LessEq},[e_2]) \\
[e_1 = e_2] &= A.\text{Compare}([e_1],A.\text{Equal},[e_2]) \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 &= A.\text{If}([e_1],[e_2],[e_3]) \\
\text{let } d_1 \cdots d_n \text{ in } e &= A.\text{Let}([d_1],\ldots,[d_n],[e]) \\
\text{fn } x : t \Rightarrow e &= A.\text{Fn}("x",[t],[e]) \\
[e_1 \cdot e_2] &= A.\text{Apply}([e_1],[e_2]) \\
(e_1,e_2) &= A.\text{Pair}([e_1],[e_2]) \\
#1 e &= A.\text{Fst}([e]) \\
#2 e &= A.\text{Snd}([e]) \\
\text{val } x : t = e &= A.\text{Val}_d("x",[t],[e]) \\
\text{fun } f(x:t_1):t_2 = e_2 &= A.\text{Fun}_d("f","x",[t_1],[t_2],[e_2])
\end{align*}\]

Figure 3: Datatype Representation of Syntax