MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Problem Set 6

Problem 1: `cellof` Subtyping

Do exercise 11.6 on page 400 of the course notes.

Problem 2: Polymorphic Types

Below are some programs written in Scheme/Y. Translate them into Scheme/XSP, being as faithful as possible in maintaining the spirit of the original computation.

```
a. (program
     (define positive? (lambda (n) (primop > n 0)))
     (define compose (lambda (f g) (lambda (x) (f (g x)))))
     ((compose not? positive?) 3))
```

```
b. (program
     (define sum-integers
       (lambda (int-list)
         (if (null? int-list)
             0
             (if (primop integer? (primop car int-list))
                 (primop + (primop car int-list)
                           (sum-integers (primop cdr int-list)))
                 (sum-integers (primop cdr int-list))))))
     (sum-integers (list 1 (symbol a) 2 #t)))
```

```
c. (program
     (define inc (lambda (n) (primop + n 1)))
     (define twice (lambda (f) (lambda (x) (f (f x)))))
     (((twice twice) inc) 3))
```

Problem 3: Exception Types

Alyssa P. Hacker decides to extend Scheme/X with the dynamic exception handling primitives `catch` and `throw`:

$$E ::= (\texttt{throw}\ E_v)\ |\ (\texttt{catch}\ E_h\ E_b)\ |\ \dots$$

The informal operational semantics of these constructs is as follows:

- (throw E_v) signals a dynamic exception whose value V is the value of E_v. When an exception is signalled via `throw`, normal evaluation is aborted, and the first *dynamically* enclosing `catch` is found to handle the exception. If there is no enclosing `catch`, the behavior of `throw` is undefined.

- (catch E_h E_b) first evaluates E_b. If an exception with value V is signalled during the evaluation of E_b, the value of the catch expression is the value of (call E_h V). If no exception is signalled during the evaluation of E_b, the value of the catch expression is the value of E_b. Note that the handler expression E_h is never evaluated if no exception is raised in the body E_b.

For example:

```
(let ((foo (lambda ((x int))
              (if (= x 0)
                  (throw 300)
                  1000))))
  (catch (lambda ((x int)) (+ x 1))
    (+ 20 (foo 7))))
⇒ 1020

(let ((foo (lambda ((x int))
              (if (= x 0)
                  (throw 300)
                  1000))))
  (catch (lambda ((x int)) (+ x 1))
    (+ 20 (foo 0))))
⇒ 301

(catch (lambda ((x int)) (+ x 1))
  (throw (throw 7)))
⇒ 8
```

Alyssa also modifies the type system of Scheme/X. She assigns each expression two types, a normal type (denoted with :) and an exception type (denoted with $). When she writes

$$E : T_n \$ T_e$$

she means that expression E has normal type T_n and exceptional type T_e. The exception type is the type of the argument to the throw that raised the exception. An expression that cannot produce an exceptional (normal) value will have void as its exceptional (normal) type. Here are some examples:

$$(\texttt{throw 1}) : \texttt{void} \, \$ \, \texttt{int}$$
$$1 : \texttt{int} \, \$ \, \texttt{void}$$
$$(\texttt{if \#t (throw (symbol true)) 1}) : \texttt{int} \, \$ \, \texttt{sym}$$
$$(\texttt{throw (throw 1)}) : \texttt{void} \, \$ \, \texttt{int}$$
$$(\texttt{catch (lambda ((x int)) x) (throw 1)}) : \texttt{int} \, \$ \, \texttt{void}$$

Notice that an exception type is not needed for the arguments to a procedure. Also notice that the phrase $T_n \$ T_e$ is *not* a type.

The Scheme/X typing operator : is a relation on Expression × Type, while Alyssa's new typing operator : $ is a relation on Expression × Type × Type. Type environments are the same for both relations: they map identifiers to types. In particular, in the case of : $, type environments do *not* map identifiers to pairs of types.

Alyssa also wants to add a limited form of subtyping to the language. (Recall that Alyssa is starting with Scheme/X and *not* Scheme/XS, so that there is no subtyping in the original language). She decides that void is a subtype of all types, and this is the only subtyping relationship that she allows in the language. She defines *LUB*, a least-upper-bound operator that takes a sequence of types and has the following functionality: if all of the types in the sequence are void, *LUB* returns void; if all the non-void types are the same type, *LUB* returns this type; otherwise, *LUB* is undefined.

For example:

$$LUB \; [\texttt{void, void}] \;\; = \texttt{void}$$
$$LUB \; [\texttt{int, int}] \;\; = \texttt{int}$$
$$LUB \; [\texttt{void, int}] \;\; = \texttt{int}$$
$$LUB \; [\texttt{bool, void, bool}] \;\; = \texttt{bool}$$
$$LUB \; [\texttt{void, bool, int}] \;\; = undefined$$

Here is how *LUB* is used in Alyssa's rule for `if`:

$$
\frac{
\begin{array}{c}
A \vdash E_1 : T_{n_1} \; \$ \; T_{e_1} \\
A \vdash E_2 : T_{n_2} \; \$ \; T_{e_2} \\
A \vdash E_3 : T_{n_3} \; \$ \; T_{e_3} \\
T_{n_1} \sqsubseteq \texttt{bool} \\
T_n = LUB \; [T_{n_2}, T_{n_3}] \\
T_e = LUB \; [T_{e_1}, T_{e_2}, T_{e_3}]
\end{array}
}{
A \vdash (\texttt{if} \; E_1 \; E_2 \; E_3) : T_n \; \$ \; T_e
} \quad [\textit{if}]
$$

Unfortunately, Alyssa was called away to help with 6.001 before she could complete her type checking rules and you are asked to help out.

a. Modify the grammar of Scheme/X types from Figure 11.4 on page 375 of the course notes to accommodate Alyssa's new features.

b. Give the typing rules for each of the following forms:

  (i) `throw`

  (ii) `catch`

  (iii) `lambda`

  (iv) application

c. Give a modified subtyping rule for procedure types.

Problem 4: Type Reconstruction

In this problem, you will extend the type system of Scheme/R to support the `module` and `with` forms. (Scheme/R is presented in section 11.7 of the course notes.)

The expression and type grammars of Scheme/R are extended as follows:

$$
\begin{array}{lll}
E & ::= & (\texttt{module} \; (\texttt{define} \; I \; E)^*) \; | \; (\texttt{with} \; (I^*) \; E_m \; E_b) \; | \; \ldots \\
T & ::= & (\texttt{moduleof} \; (\texttt{val} \; I \; T)^*) \; | \; \ldots
\end{array}
$$

The (I^*) that appears in the `with` expression is the list of identifiers which are defined by the module value of E_m.

There are many examples of the use of `module` and `with` throughout the course notes. We will just give one example here to illustrate how we would like you to type them. The expression

```
(let ((m (module (define id (lambda (x) x)))))
   (with (id) m (if (id #f) (id 3) (id 4))))
```

should be well-typed, and have type `int`. Note in particular the polymorphic use of `id` in the body of the `with` expression.

a. Extend the typing rules of Scheme/R (given in Figure 11.15 of the course notes) to handle `module` and `with`.

b. Suppose the syntax of `with` were changed to be

$$(\text{with } E_m \ \ E_b).$$

Briefly explain how this would prevent you from providing a working type reconstruction algorithm. You should provide a relevant example input expression.

c. Extend the type reconstruction algorithm of Scheme/R (presented in Figures 11.16 and 11.17 of the course notes) to handle `module` and `with`.

d. The file `code/ps6/recon.scm` in the course directory contains an implementation of the Scheme/R type reconstruction algorithm.

Extend `recon.scm` to handle `module` and `with`. We have written `recon.scm` so that you only need to define two functions, `reconstruct-module` and `reconstruct-with`, in order to handle `module` and `with`.

An appendix documenting `recon.scm` is attached to this problem set.

e. Provide a transcript showing that your implementation works. Your transcript must show how your implementation handles the following test cases (which can be found in the file `module-examples.scm`):

```
(let ((m (module
           (define twice (lambda (f)
                           (lambda (x)
                             (f (f x)))))))
  (with (twice) m
    (if ((twice not?) #f)
        ((twice (lambda (x) (+ 1 x))) 4)
        5)))

(let ((m (module
           (define a 4)
           (define b 5)))
      (b 6))
  (with (a) m b))
```

You should also run your code on other test cases of your own choosing. Some of these should be well-typed using your rules, and some should not be well-typed.

Documentation for `recon.scm`

This appendix contains some information about the file `recon.scm`, which is an implementation of the Scheme/R type reconstruction algorithm described in Section 11.7 of the course notes.

Before extending the type reconstruction program, we suggest that you first play with the existing implementation. The file `recon-test.scm` contains a number of interesting Scheme/R expressions to type-check. You should also invent some examples of your own to help you understand the power and limitations of type reconstruction.

The file `module-examples.scm` contains some expressions that you should test your extensions on. You should also invent some test cases of your own.

Beware that `recon.scm` *is written in a different style than the algorithm R of the course notes.* `recon.scm` uses side effects to perform unification on a global substitution, and so a substitution is not passed explicitly as an argument to the reconstruction algorithm.

Expressions

The Scheme+ datatype `exp` describes the grammar of Scheme/R expressions *including* `module` *and* `with`:

```
(define-datatype exp
  (unit->exp)
  (boolean->exp bool)
  (integer->exp int)
  (string->exp string)
  (symbol->exp sym)
  (variable->exp sym)
  (lambda->exp (listof sym) exp)
  (call->exp exp (listof exp))
  (if->exp exp exp exp)
  (primop->exp primop (listof exp))
  (let->exp (listof definition) exp)
  (letrec->exp (listof definition) exp)
  (set!->exp sym exp)
  (begin->exp exp exp)
  (module->exp (listof definition))
  (with->exp (listof sym) exp exp)
  )

(define-datatype definition
  (make-definition sym exp))
```

Type Expressions and Type Schemas

The Scheme+ datatype `type` describes the grammar of type expressions. The base type case is used for all base types, while the `compound` case is used for compound types (currently this includes procedure types and list types, but it could perhaps be extended to include tagged union types and reference types as well). The `moduleof` case is used to describe the types of module values.

```
(define-datatype type
  ;; type variable
  (tvariable->type tvariable)
  ;; (unit, bool, int, string, symbol)
  (base->type sym)
  ;; ->, list-of, etc.
  (compound->type sym (listof type))
  ;; placeholder for unconstrained tvars
```

```
(unknown->type)
;; (moduleof (val I T)*)
(moduleof->type (listof sym) (listof type))
)
```

A number of base types are defined, as well as the constructor `make-arrow-type`, which takes a list of argument types and the result type and returns the corresponding procedure type.

```
boolean-type : type
integer-type : type
string-type  : type
symbol-type  : type
unit-type    : type

make-arrow-type : (-> ((listof type) type) type)
```

The function `new-tvariable` creates a fresh type variable:

```
new-tvariable : (-> (symbol) tvariable)
```

The argument of `new-tvariable` is only for identification purposes; it has no semantic content.

The `unparse-type` procedure is handy for debugging:

```
unparse-type : (-> (type) sexp)
```

Type schemas are *not* types. The following datatype is useful (in type environment manipulations) for determining whether a variable is bound to a type variable or a schema:

```
(define-datatype tvar-or-schema
  (tvar->tvar-or-schema tvariable)
  (schema->tvar-or-schema schema))

(define-datatype schema
  (make-schema (listof tvariable) type))
```

The function `compute-schema` creates a type schema from a type and an environment, using the current substitution. It corresponds to the function *Rgen* in the reconstruction algorithm. The function `instantiate-schema` creates an instance of the schema by replacing the bound variables by fresh type variables.

```
; (compute-scheme T A) = Rgen(T, A, S), with S implicit
compute-schema : (-> (type tenv) schema)
instantiate-schema : (-> (schema) type)
```

Unification

The `unify!` procedure unifies two types and returns unit. *Unlike the unification function in the course notes*, this is a side-effecting version of unification; it does not return the resulting substitution, but simply mutates the current substitution to be the result of the unification. A mutation-based version of unification makes sense because the substitution as used in the reconstruction algorithm is single-threaded, like a store. `unify!` generates an error if the expressions don't unify.

```
unify! : (-> (type type) unit)
```

Type Environments

A type environment or type assignment maintains bindings between identifiers and either type variables or type schemas. The interface to type environments is defined by these functions:

```
tlookup : (-> (tenv var) tvar-or-schema)
extend-by-tvariables : (-> (tenv (list-of symbol)
                                (list-of tvariable))
                           tenv)
extend-by-schemas : (-> (tenv (list-of symbol)
                              (list-of schema))
                        tenv)
```

Top Level

The top level call to the type reconstruction algorithm is through the procedure recon. This procedure takes an s-expression as an argument, parses it, performs type reconstruction, and returns the unparsed type as an s-expression. It signals an error if the type cannot be reconstructed.

```
recon: (-> (sexp) sexp)
```

The reconstruct procedure takes an expression and a type environment, and returns the type of the expression in that type environment. (It signals an error if the type cannot be reconstructed.) This is the main dispatch for the type reconstruction algorithm. It handles literals directly, and otherwise dispatches to the appropriate specialist routine reconstruct-variable, etc.

```
(define (reconstruct exp tenv)
  (match exp
    ((unit->exp)      unit-type)
    ((boolean->exp _) boolean-type)
    ((integer->exp _) integer-type)
    ((string->exp _)  string-type)
    ((symbol->exp _)  symbol-type)
    ((variable->exp var)
     (reconstruct-variable var tenv))
    ((lambda->exp formals body)
     (reconstruct-lambda formals body tenv))
    ((call->exp op args)
     (reconstruct-call op args tenv))
    ((if->exp test con alt)
     (reconstruct-if test con alt tenv))
    ((let->exp defs body)
     (reconstruct-let defs body tenv))
    ((letrec->exp defs body)
     (reconstruct-letrec defs body tenv))
    ((module->exp defs)
     (reconstruct-module defs tenv))      ;*
    ((with->exp vars mod body)
     (reconstruct-with vars mod body tenv)) ;*
    ))
```

reconstruct is essentially the function R described in the type reconstruction algorithm, except that the current substitution is passed implicitly (using side effects). The procedures used in the two *'ed lines are not provided; you must write these as a part of your assignment.