Programming Language Concepts

CSCI-344 **Recitation 7**
Term 20161

Polymorphic Type Systems (Typed μScheme)

1 Introduction

You will undertake a few simple problems in order to develop basic skills with a polymorphic type system.

2 Description

- (Adapted from Exercise 14 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 453).)

  Recall the polymorphic `map` function in Typed μScheme:

```
(val-rec
   (forall ['a 'b] (('a -> 'b) (list 'a) -> (list 'b)))
   map
   (type-lambda ['a 'b]
      (lambda ([f : ('a -> 'b)] [xs : (list 'a)])
         (if ((@ null? 'a) xs)
             (@ '() 'b)
             ((@ cons 'b) (f ((@ car 'a) xs)) ((@ map 'a 'b) f ((@ cdr 'a) xs)))))))
```

  It may be easier to understand the definition by using a `let` expression to perform all of the type applications:

```
(val-rec
   (forall ['a 'b] (('a -> 'b) (list 'a) -> (list 'b)))
   map
   (type-lambda ['a 'b]
      (lambda ([f : ('a -> 'b)] [xs : (list 'a)])
         (let ((null?@'a  (@ null? 'a))
               (car@'a    (@ car 'a))
               (cdr@'a    (@ cdr 'a))
               (nil@'b    (@ '() 'b))
               (cons@'b   (@ cons 'b))
               (map@'a-'b (@ map 'a 'b)))
              (if (null?@'a xs)
                  nil@'b
                  (cons@'b (f (car@'a xs)) (map@'a-'b f (cdr@'a xs))))))))
```

  Implement `exists?` and `all?` in Typed μScheme.

  – Give the polymorphic type of both functions.
  Hint: Both functions should have the same type.

  – Write the function `exists?` (using `val-rec`).

  – Write the function `all?` (using `val`, `exists?`, and De Morgan's laws).

- (Adapted from Exercise 35 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 462) (itself adapted from an earlier version of this recitation problem!).)

  The typing rule for TYLAMBDA/`type-lambda` may seem complicated with the side condition $\alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n$. The purpose of this problem is to demonstrate that a simpler typing rule would be *unsound*, which is to say that it would not actually guarantee the safety property that it is meant to enforce. (In fact, an earlier draft of *Programming Languages: Build, Prove, and Compare* used this simpler, but unsound, rule.)

  An earlier draft of *Programming Languages: Build, Prove, and Compare* used the following typing rule for TYLAMBDA/`type-lambda` (written using concrete Typed μScheme syntax):

$$\frac{\Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \texttt{(type-lambda (}\alpha_1 \cdots \alpha_n\texttt{) } e\texttt{)} : \texttt{(forall (}\alpha_1 \cdots \alpha_n\texttt{) } \tau\texttt{)}} \text{ (TYLAMBDA(OLD))}$$

  and the current draft of *Programming Languages: Build, Prove, and Compare* uses the following typing rule for TYLAMBDA/`type-lambda`:

$$\frac{\alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n \qquad \Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \texttt{(type-lambda (}\alpha_1 \cdots \alpha_n\texttt{) } e\texttt{)} : \texttt{(forall (}\alpha_1 \cdots \alpha_n\texttt{) } \tau\texttt{)}} \text{ (TYLAMBDA(NEW))}$$

  where $\text{frv}(\Gamma)$ is the set of *free type variables* of Γ.

  Consider the following Typed μScheme program (sequence of definitions):

```
(val cast
   (type-lambda ['a 'b]
      (lambda ([x : 'a])
         (let ((y (type-lambda ['a] x)))
            (@ y 'b)))))

(val cast@int-fn (@ cast int (int -> int)))

(val f (cast@int-fn 42))

(val ans (f 0))
```

  - Does this program "go wrong" when evaluated? Why or why not?
    Hint: Imagine the program with all `(type-lambda (`$\alpha_1 \cdots \alpha_n$`) e)` expressions replaced by e and with all `(@ e` $\tau_1 \cdots \tau_n$`)` expressions replaced by e, since evaluation of Typed μScheme behaves exactly as if these constructs aren't there.
  - Is this program well-typed using TYLAMBDA(OLD)? Why or why not?
    Hint: Determine the environments Δ and Γ under which `(type-lambda ['a] x)` is type checked.
    Hint: Determine the types of `y`, `(@ y 'b)`, and `cast`.
  - Is this program well-typed using TYLAMBDA(NEW)? Why or why not?
    Hint: Determine the environments Δ and Γ under which `(type-lambda ['a] x)` is type checked.

- One of the great advantages of a polymorphic type system with kinds and quantified types is that it admits the expression of user-defined polymorphic abstract datatypes, allowing programmers (and not just language designers/implementers) to add a new type constructor and operations for that type constructor without changing the abstract syntax, the values, the type checker, or the evaluator for the language. This problem explores one approach to user-defined polymorphic abstract datatypes. See Exercises 24, 25, 26, 28, and 29 of Chapter 6 *Programming Languages: Build, Prove, and Compare* for other approaches.

  One way to introduce a polymorphic abstract datatype is to introduce one or more type constructors (with declared kinds) and one or more variables (with declared types) and to give definitions that provide implementations for the variables. In order to keep things simple, the definitions that provide implementations for the variables are *unchecked*, meaning that they are not type checked and are assumed to be correct with respect to the declared types of the variables.

  To support this approach, we imagine extending Typed μScheme with a new kind of definition: `abstype`. Here is an example that introduces a polymorphic abstract datatype for binary trees:

```
(abstype
   ;; declared type constructors with kinds
   ([btree  : (* => *)])
   ;; declared variables with types
   (;; creators/producers // introduction
    [leaf   : (forall ['a] (btree 'a))]
    [node   : (forall ['a] ((btree 'a) 'a (btree 'a) -> (btree 'a)))]
    ;; observers // elimination
    [leaf?  : (forall ['a] ((btree 'a) -> bool))]
    [node?  : (forall ['a] ((btree 'a) -> bool))]
    [node-l : (forall ['a] ((btree 'a) -> (btree 'a)))]
    [node-x : (forall ['a] ((btree 'a) -> 'a))]
    [node-r : (forall ['a] ((btree 'a) -> (btree 'a)))])
   ;; definitions for implementation (unchecked)
   (;; creators/producers // introduction
    (val leaf   (list1 'leaf))
    (val node   (lambda ([l : _ l] [x : _] [r : _]) (list4 'node l x r)))
    ;; observers // elimination
    (val leaf?  (lambda ([t : _]) (= (car t) 'leaf)))
    (val node?  (lambda ([t : _]) (= (car t) 'node)))
    (val node-l cadr)
    (val node-x caddr)
    (val node-r cadddr)))
```

  When elaborating (i.e., type checking) this `abstype` definition, the kind environment Δ is extended by adding the type constructor `btree` with kind `(* => *)` and the type environment Γ is extended by adding the variables `leaf` with type `(forall ['a] (btree 'a))`, `node` with type `(forall ['a] ((btree 'a) 'a (btree 'a) -> (btree 'a)))`, ..., and `node-r` with type `(forall ['a] ((btree 'a) -> (btree 'a)))`; however, the definitions for implementation are not elaborated (i.e., type checked). On the other hand, when evaluating this `abstype` definition, the value environment ρ is extended by evaluating the definitions for implementation. Note that the definitions for implementation are written in the syntax of Typed μScheme; in particular, explicit types are required for parameters of `lambda` abstractions, but, since they will not be used for type checking, a dummy symbol is used. (We choose to adopt the syntax of Typed μScheme for the (unchecked) definitions in order to reuse the existing parsing infrastructure.) Also note that the definitions for implementation could not be type checked as Typed μScheme (even with the addition of `type-lambda` and `@` expressions); in particular, the list created by the `node` function is *heterogenous* (containing elements of different types).

  Although (checked) run-time errors can occur when the `node-l` or `node-x` or `node-r` function is applied to the `leaf` binary tree, no run-time errors can occur when the `leaf?` or `node?` function is applied to a value, because the declared types ensure that `leaf?` and `node?` can only be applied to values produced by `leaf` and `node`, which are necessarily non-empty lists with a symbol as the first element.

  One way to think about the `abstype` definition is that it provides a statically-typed interface to an untyped implementation. The typed interface is carefully constructed so that clients cannot violate type soundness by using the exported values.

Implement product and sum types in Typed μScheme using `abstype`.

- Give an `abstype` definition for the `prod` type constructor and the `pair`, `fst`, and `snd` polymorphic functions.
  Hint: Despite the long setup, the interface and implementation of products is quite short. For the implementation, consider how you would implement products in (untyped) μScheme.
- Give an `abstype` definition for the `sum` type constructor and the `left`, `right`, and `either` polymorphic functions.
  Hint: The interface and implementation of sums is a little more complex than that of products. For the implementation, consider how you would implement sums in (untyped) μScheme.

3 Requirements and Submission

At the end of class, submit the group's solutions as hard-copy; be sure to include the names of all group members in the submission.

A Interpreter

A reference Typed μScheme interpreter is available on the CS Department Linux systems (e.g., `glados.cs.rit.edu` and `queeg.cs.rit.edu` and ICLs 1 and 2) at:

`/usr/local/pub/mtf/plc/bin/tuscheme`

Use the reference interpreter to check your code.

Solutions

- Typed μScheme functions
  - `exists?`

```
(val-rec
   (forall ['a] (('a -> bool) (list 'a) -> bool))
   exists?
   (type-lambda ['a]
      (lambda ([f : ('a -> bool)] [xs : (list 'a)])
         (let ((null?@'a   (@ null? 'a))
               (car@'a     (@ car 'a))
               (cdr@'a     (@ cdr 'a))
               (exists?@'a (@ exists? 'a)))
              (if (null?@'a xs)
                  #f
                  (if (f (car@'a xs))
                      #t
                      (exists?@'a f (cdr@'a xs)))))))))
```

  - `all?`

```
(val
   all?
   (type-lambda ['a]
      (lambda ([f : ('a -> bool)] [xs : (list 'a)])
         (let ((not-f (lambda ([x : 'a]) (not (f x)))))
            (not ((@ exists? 'a) not-f xs)))))))
```

- (un)soundness of Typed μScheme
  - goes wrong: "applied non-function"; consider the corresponding (untyped) μScheme program:

```
(val cast
   (lambda (x)
      (let ((y x))
           y)))

(val cast@int-fn cast)

(val f (cast@int-fn 42))

(val ans (f 0))
```

    `cast` is a glorified identity function and, therefore, `f` is bound to the value `42`

  - is well-typed; the environments under which `(type-lambda ['a] x)` is type checked are:

$$\Delta = \Delta_0\{\texttt{'a} :: *, \texttt{'b} :: *\} \qquad \text{and} \qquad \Gamma = \Gamma_0\{\texttt{x} \mapsto \texttt{'a}\}$$

    and we can derive

$$\dfrac{\dfrac{(\Gamma_0\{\texttt{x} \mapsto \texttt{'a}\})(\texttt{x}) = \texttt{'a}}{\Delta_0\{\texttt{'a} :: *, \texttt{'b} :: *\}\{\texttt{'a} :: *\}, \Gamma_0\{\texttt{x} \mapsto \texttt{'a}\} \vdash \texttt{x} : \texttt{'a}}\ (\textsc{Var})}{\Delta_0\{\texttt{'a} :: *, \texttt{'b} :: *\}, \Gamma_0\{\texttt{x} \mapsto \texttt{'a}\} \vdash \texttt{(type-lambda ['a] x)} : \texttt{(forall ['a] 'a)}}\ (\textsc{Tylambda(Old)})$$

    Therefore, `y` has the type `(forall ['a] 'a)`, `(@ y 'b)` has the type `'b`, and `cast` has the type `(forall ['a 'b] ('a -> 'b))`.

    The "problem" is that the type of `x` in the environment $\Gamma_0\{\texttt{x} \mapsto \texttt{'a}\}$ uses a type variable `'a` that was introduced by the outer `type-lambda` but is also being introduced by the inner `type-lambda`. Thus, the inner `type-lambda` is *capturing* the type variable `'a` used in the type of `x`, which results in a misinterpretation of the type of `x` and unsoundness.

  - is not well-typed; the environments under which `(type-lambda ['a] x)` is type checked are:

$$\Delta = \Delta_0\{\texttt{'a} :: *, \texttt{'b} :: *\} \qquad \text{and} \qquad \Gamma = \Gamma_0\{\texttt{x} \mapsto \texttt{'a}\}$$

    and we cannot derive

$$\dfrac{\texttt{'a} \notin \text{ftv}(\Gamma_0\{\texttt{x} \mapsto \texttt{'a}\}) \qquad \dfrac{(\Gamma_0\{\texttt{x} \mapsto \texttt{'a}\})(\texttt{x}) = \texttt{'a}}{\Delta_0\{\texttt{'a} :: *, \texttt{'b} :: *\}\{\texttt{'a} :: *\}, \Gamma_0\{\texttt{x} \mapsto \texttt{'a}\} \vdash \texttt{x} : \texttt{'a}}\ (\textsc{Var})}{\Delta_0\{\texttt{'a} :: *, \texttt{'b} :: *\}, \Gamma_0\{\texttt{x} \mapsto \texttt{'a}\} \vdash \texttt{(type-lambda ('a) x)} : \texttt{(forall ('a) 'a)}}\ (\textsc{Tylambda(New)})$$

    because $\texttt{'a} \notin \text{ftv}(\Gamma_0\{\texttt{x} \mapsto \texttt{'a}\})$ is not true.

Note that we can make the `cast` function well-typed (using both Tylambda(Old) and Tylambda(New)) by choosing a distinct type variable for the inner `type-lambda`:

```
(val cast
   (type-lambda ('a 'b)
      (lambda ([x : 'a])
         (let ((y (type-lambda ['c] x)))
            (@ y 'b)))))
(val cast@int-fn (@ cast int (int -> int)))
(val f (cast@int-fn 42))
(val ans (f 0))
```

Now, `y` has the type `(forall ['c] 'a)`, `(@ y 'b)` has the type `'a`, and `cast` has the type `(forall ['a 'b] ('a -> 'a))`. Furthermore, `cast@int-fn` has the type `(int -> int)` and `f` has the type `int`. However, the expression `(f 0)` is ill-typed.

- polymorphic abstract types (with unchecked implementations)
  - `prod` type constructor; `pair`, `fst`, `snd` polymorphic functions

```
(abstype
   ;; declared type constructors with kinds
   ([prod : (* => * => *)])
   ;; declared variables with types
   ([pair : (forall ['a 'b] ('a 'b -> (prod 'a 'b))) pair]
    [fst  : (forall ['a 'b] ((prod 'a 'b) -> 'a)) fst]
    [snd  : (forall ['a 'b] ((prod 'a 'b) -> 'b)) snd])
   ;; definitions for implementation (unchecked)
   ((val pair cons)
    (val fst car)
    (val snd cdr)))
```

  - `sum` type constructor; `left`, `right`, `either` polymorphic functions

```
(abstype
   ;; declared type constructors with kinds
   ([sum    : (* => * => *) sum])
   ;; declared variables with types
   ([left   : (forall ['a 'b] ('a -> (sum 'a 'b)))]
    [right  : (forall ['a 'b] ('b -> (sum 'a 'b)))]
    [either : (forall ['a 'b 'c] ((sum 'a 'b) ('a -> 'c) ('b -> 'c)))])
   ;; definitions for implementation (unchecked)
   ((val left   (lambda ([x : _]) (cons #t x)))
    (val right  (lamdba ([x : _]) (cons #f x)))
    (val either (lambda ([s : _] [f : _] [g : _])
                  (if (car s) (f (cdr s)) (g (cdr s)))))))
```