Advance Programming Languages  Assignment 5

CS 6110 Spring 2010  Due Monday, 3 May 2010

---

*As before, you may work alone or in groups of two or three. If you work in a group, please form a group in CMS. Only one person needs to submit the assignment.*

1. Denesting Loops

   Here is a basic loop-denesting transformation that an optimizing compiler might perform:

```
                                if b then {
   while b do {                    p;
      p;                 ⇒         while b ∨ c do {
      while c do q                    if c then q else p
   }                               }
                                }
```

   Assume that the tests $b$ and $c$ have no side effects. Prove by equational reasoning in KAT (Lecture 18) that this transformation is correct.

2. Java Bytecodes

   Consider the following syntax, which describes a small subset of the bytecode instructions in Java:

$$\begin{array}{rcl} s & ::= & \mathsf{bipush}\ k \mid \mathsf{iload}\ m \mid \mathsf{istore}\ m \mid \mathsf{iadd} \mid \mathsf{ifeq}\ i \\ & & \mathsf{new} \mid \mathsf{aload}\ m \mid \mathsf{astore}\ m \mid \mathsf{if\_acmpeq}\ i \end{array}$$

   where $k, m, i \in \mathbb{N}$. A *program* is a finite sequence $s_0, \ldots, s_{n-1}$ of such instructions.

   Values in this language are integers $k$ and object references $\ell$. Instructions are executed on an abstract stack machine. The machine has a finite set of typed variables and a stack. Informally, the instructions push integer constants on the stack (bipush), create new objects (new), transfer integers and object references between variables and the stack (iload, istore, aload, astore), or perform arithmetic (iadd). The control-flow constructs include branch instructions that test integers for zero (ifeq) or compare two references (if_acmpeq).

   In the operational semantics of this language, a configuration $\langle i, \sigma, \rho \rangle$ consists of a program counter $i$, where $0 \le i \le n$, a stack $\sigma$ of values, and an environment $\rho$ that maps variables to values. We model variables as integers $m$, so $\rho$ is a map from integers to values. The first component of a configuration indicates the instruction to execute next. A *halt configuration* is one whose first component is $n$, the number of instructions in the program.

   The operational semantics is specified by the rules below. The notation $v :: \sigma$ denotes the stack with head $v$ and tail $\sigma$. We use $m$ for variables, $i, j$ for program points, $k$ for integer data, and $\ell$ for object references.

$$\frac{s_i = \mathsf{bipush}\ k}{\langle i, \sigma, \rho \rangle \to \langle i+1, k :: \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{iload}\ m \quad \rho(m) = k}{\langle i, \sigma, \rho \rangle \to \langle i+1, k :: \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{istore}\ m}{\langle i, k :: \sigma, \rho \rangle \to \langle i+1, \sigma, \rho[k/m] \rangle}$$

$$\frac{s_i = \mathsf{iadd} \quad k = k_0 + k_1}{\langle i, k_0 :: k_1 :: \sigma, \rho \rangle \to \langle i+1, k :: \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{aload}\ m \quad \rho(m) = \ell}{\langle i, \sigma, \rho \rangle \to \langle i+1, \ell :: \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{astore}\ m}{\langle i, \ell :: \sigma, \rho \rangle \to \langle i+1, \sigma, \rho[\ell/m] \rangle}$$

$$\frac{s_i = \mathsf{new} \quad \ell \text{ fresh}}{\langle i, \sigma, \rho \rangle \to \langle i+1, \ell :: \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{ifeq}\ j \quad k = 0}{\langle i, k :: \sigma, \rho \rangle \to \langle j, \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{ifeq}\ j \quad k \neq 0}{\langle i, k :: \sigma, \rho \rangle \to \langle i+1, \sigma, \rho \rangle}$$

$$\frac{s_i = \mathsf{if\_acmpeq}\ j \quad \ell_0 = \ell_1}{\langle i, \ell_0 :: \ell_1 :: \sigma, \rho \rangle \to \langle j, \sigma, \rho \rangle} \qquad \frac{s_i = \mathsf{if\_acmpeq}\ j \quad \ell_0 \neq \ell_1}{\langle i, \ell_0 :: \ell_1 :: \sigma, \rho \rangle \to \langle i+1, \sigma, \rho \rangle}$$

We want to derive typing rules for these bytecode instructions. These rules will involve a type environment $\Gamma : \mathbb{N} \to \{\mathsf{int}, \mathsf{ref}\}$ for variables. The type of variable $m$ is $\Gamma(m)$.

*Stack types* $\tau \in \mathit{SType}$ are defined by the following grammar:

$$\tau \quad ::= \quad \mathsf{empty} \;\mid\; \alpha \;\mid\; \mathsf{int} :: \tau \;\mid\; \mathsf{ref} :: \tau$$

Here $\mathsf{empty}$ represents the empty stack, and $\alpha$ is a type variable that refers to any stack type, including $\mathsf{empty}$. Thus a stack type consists of either an empty stack or a stack variable with a finite number of specific primitive types $\mathsf{int}$ or $\mathsf{ref}$ at the top.

The typing rules for each instruction $s$ are of the form $\Gamma \vdash s : \tau_1 \to \tau_2$, meaning that starting in a state where the stack has type $\tau_1$ and variable types are given by $\Gamma$, the execution of $s$ will produce a new stack with type $\tau_2$. Stack types containing a type variable $\alpha$ are considered to be polymorphic, meaning that if $\Gamma \vdash s : \tau_1 \to \tau_2$ holds, then it should also hold for any substitution instance.

In real Java systems, the bytecode verifier infers a stack type $\Delta(i)$ at each program point $i$, $0 \le i \le n$. Once this function is defined, one can typecheck a program by checking if $\Delta$ correctly characterizes the possible types at each point. The condition to be verified is:

$$\forall i, j \quad \Gamma \vdash s_i : \Delta(i) \to \tau \;\wedge\; \mathrm{flow}(i, j) \quad \Rightarrow \quad \tau \le \Delta(j),$$

where $\mathrm{flow}(i, j)$ indicates that there is flow of control from $i$ to $j$: either $j = i + 1$ or instruction $s_i$ is a conditional jump with target $j$, and $\tau_1 \le \tau_2$ means that $\tau_1$ is more specific than (i.e., is a substitution instance of) $\tau_2$; for instance, $\mathsf{int} :: \mathsf{ref} :: \alpha \le \mathsf{int} :: \alpha$.

(a) Write the typing rules $\Gamma \vdash s : \sigma \to \tau$ for each of the bytecode instructions $s$ listed above.

(b) Describe how a suitable type annotation $\Delta : \{0, 1, \ldots, n\} \to \mathit{SType}$ might be computed, starting from some given start configuration $\langle 0, [\,], \rho \rangle$.

(c) Define satisfiability relations $\sigma \vDash \tau$ between stacks and stack types and $\rho \vDash \Gamma$ between variable environments and type environments. Using these relations, give a precise formulation of operational soundness (Lecture 26) in this framework.

3. Implementing Let-Polymorphism

In this programming exercise, you will implement type inference for let-polymorphism as described in Lecture 30. This is the kind of polymorphism on which the type systems of Standard ML, OCaml, and Haskell are based.

We have augmented our FL implementation from Assignment 3 with some extra infrastructure for type inference, but there are some missing pieces that you will have to fill in.

(a) Implement the four missing clauses in `Typecheck.typecheck` for `Let`, `Letrec`, `App` (function application), and `Cond` (conditional test).

(b) Make `Type.unify` perform an occurs check. Right now attempted type unifications of the form $\alpha = \alpha \to \beta$ for example are not caught.

For (a), you will find all the basic infrastructure you need in the `Type` module. All you have to do is figure out what subroutines to call in what order.

The method used for unification is slightly different from the method presented in class. It uses references for variables for ease of substitution. An invariant of the unification process is that there is only one occurrence of each variable; thus substitution of another term for the variable has immediate global effect.

Recall from Lecture 30 that the let rule is

$$\frac{\Gamma \vdash d : \sigma \qquad \Gamma,\, x : \forall\alpha_1 \ldots \forall\alpha_n . \sigma \vdash e : \tau \qquad \{\alpha_1, \ldots, \alpha_n\} = \mathit{FV}(\sigma) - \mathit{FV}(\Gamma)}{\Gamma \vdash \mathsf{let}\ x = d\ \mathsf{in}\ e : \tau}$$

The `Let` clause should implement this rule. Thus to typecheck a let statement $\mathsf{let}\ x = d\ \mathsf{in}\ e$, you should first typecheck $d$ in the supplied environment $\Gamma$ to obtain $\sigma$, then append a new type judgment

$x : \forall\alpha_1 \ldots \forall\alpha_n.\sigma$ to $\Gamma$ and typecheck $e$ in that environment, where $\{\alpha_1, \ldots, \alpha_n\}$ is the set of type variables occurring in $\sigma$ that do not occur free in $\Gamma$. The resulting type is the type of let $x = d$ in $e$.

The polymorphic type $\forall\alpha_1 \ldots \forall\alpha_n.\sigma$ is called a *type schema*. A type environment $\Gamma$ is a mapping from type variables to type schemas (not types). We represent schemas as a pair consisting of a list of quantified variables and a quantifier-free type:

```
  type schema = id list * typ
```

Note that the `Var x` clause looks up the value of the variable in the type environment, which will be a schema; then it instantiates all the quantified variables in the schema with fresh type variables. This amounts to the application of the instantiation ($\forall$-elimination) rule.

```
  | Var x -> instantiate (lookup x gamma)
```

`Letrec` is similar to `Let`, except that in the first step, $d$ should be typechecked in the environment $\Gamma$ augmented with a typing premise $x : \beta$, where $\beta$ is a fresh type variable (actually, the schema $().\beta$ with no quantifiers), since $x$ may occur in $d$. The resulting type $\sigma$ should then be unified with $\beta$.

The `App` clause follows the modus ponens rule (Lecture 30, Section 3). Thus to typecheck an application $(e_0\, e_1)$, we typecheck both $e_0$ and $e_1$ in the environment $\Gamma$ to get types $\sigma_0$ and $\sigma_1$ respectively, then unify $\sigma_0$ with $\sigma_1 \to \alpha$, where $\alpha$ is a fresh type variable. The type of $(e_0\, e_1)$ is $\alpha$.

The conditional test will have to make sure that the condition is a Boolean and that the then and else clauses are the same type. This is done by unification.

Several example programs are supplied with the release for you to use to test your implementation. The following page shows a sample run with our solution code.

```
FL version 2010.1
>> load etc/s.txt
>> list
fun x -> fun y -> fun z -> ((x z) (y z))
>> type
('c -> 'e -> 'f) -> ('c -> 'e) -> 'c -> 'f
>> load etc/t1.txt
>> list
let add3 = let compose f g x = (f (g x)) in
((compose fun x -> (x + 1)) fun x -> (x + 2)) in
(add3 10)
>> run
13
>> type
int
>> load etc/omega.txt
>> list
let omega = fun x -> (x x) in
omega
>> type
occurs check violation
>> load etc/poly2.txt
>> list
let f = fun x -> x in
if (f true) then (f 3) else (f 4)
>> run
3
>> type
int
>> load etc/fact.txt
>> list
let rec fact n = if n <= 1 then 1 else (n * (fact (n - 1))) in
(fact 4)
>> run
24
>> type
int
>> load etc/basic5.txt
>> list
fun x -> (x 3)
>> type
(int -> 'ah) -> 'ah
>> load etc/poly3.txt
>> list
let rec f x y n = if n <= 1 then 1 else (n * (((f y) x) (n - 1))) in
f
>> type
'ap -> 'ap -> int -> int
>> quit
bye
```