COMP 321: Principles of Programming Languages, Spring 2014
Assignment 2: Type safety, Implementing Semantics

Out: Thursday, February 6, 2014
Due: Thursday, February 13, 2014, 2:40 PM

In this assignment, you will do a first type safety proof, and you will implement a typechecker (the static semantics) and evaluator (the dynamic semantics) for $\mathcal{L}\{\texttt{num},\texttt{str}\}$.

The written portion should be handed in in class. The programming portion should be copied to your handin directory on WesFiles.

1 Type Safety

Suppose we extended $\mathcal{L}\{\texttt{num},\texttt{str}\}$with a primitive two-binding `let`:

```
let x be 7
    y be 8
 in
    x + y
```

would evaulate to 15. `x` and `y` are both only in scope in the body of the `let`, not in each other. As an ABT, we write $\mathsf{let2}(e_1, e_2, x.y.e_3)$ for `let x be e1 y be e2 in e3`.

Here is a typing rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_1, y : \tau_2 \vdash e_3 : \tau_3}{\Gamma \vdash \mathsf{let2}(e_1, e_2, x.y.e_3) : \tau_3}\ \texttt{typing-let2}$$

Task 1 (15%). Give dynamic semantics rules for $\mathsf{let2}(e_1, e_2, x.y.e_3)$. Your semantics should evaluate the two bindings in left-to-right order, and should be call-by-value.

Task 2 (15%). Recall the progress theorem:

For all e, τ, if $\cdot \vdash e : \tau$ then either e value or there exists an e' such that $e \mapsto e'$.

Here, the notation $\cdot \vdash e : \tau$ means that e has type τ in the empty context.

Progress is proved by rule induction on the derivation of $\cdot \vdash e : \tau$. Prove the case of progress for the typing rule `typing-let2` for $\mathsf{let2}(e_1, e_2, x.y.e_3)$.

Task 3 (15%). Recall the preservation theorem:

For all e, e', τ, if $e \mapsto e'$ and $\cdot \vdash e : \tau$ then $\cdot \vdash e' : \tau$.

Preservation is proved by rule induction on $e \mapsto e'$. Prove the cases of preservation for your dynamic semantics rules from Task 1.

You may use the following lemmas:

- Inversion of typing: For all $e_1, e_2, x, y, e_3, \tau$, if $\cdot \vdash \mathsf{let2}(e_1, e_2, x.y.e_3) : \tau$ then there exist τ_1 and τ_2 such that $\cdot \vdash e_1 : \tau_1$ and $\cdot \vdash e_2 : \tau_2$ and $x : \tau_1, y : \tau_2 \vdash e_3 : \tau$.
- Weakening: For all $\Gamma, e, \tau, \tau_1, x$, if $\Gamma \vdash e : \tau$ and $x \notin \Gamma$ then $\Gamma, x : \tau_1 \vdash e : \tau$.
- Substitution: For all $\Gamma, e, x, \tau, \tau'$, if $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$ then $\Gamma \vdash [e/x]\, e' : \tau'$

2 Implementing ABTs

In homework 1, you implemented the syntax for $\mathcal{L}\{$`num`,`str`$\}$. This was a bit tricky, because you needed to remember to freshen variables (generate a new name and swap it in) at the right time. We can make programming with ABTs less error-prone by encapsulating this freshening in a separate module. The signature for such a module is as follows:

```
signature EXP =
sig
    structure N : NAME;

    (* real type of expressions *)
    type exp;

    (* we expose an exp one level as a datatype *)
    datatype exp1 =
        Var of N.name
      | Number of int
      | String of string
      | Plus of exp * exp
      | Cat of exp * exp
      | Let of exp * N.name * exp;

    (* any bound name you get back from show is fresh *)
    val show : exp -> exp1;

    val hide : exp1 -> exp;

    val equal : exp * exp -> exp;
end
```

The substructure `N` provides an implementation of names.

The representation type `exp` is hidden, so the implementation of this module can choose any type it wants. To inspect an ABT, you use the `show` function, which exposes one level of the tree as the concrete

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{num}[n] : \mathsf{num}} \qquad \frac{}{\Gamma \vdash \mathsf{str}[s] : \mathsf{str}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{num} \quad \Gamma \vdash e_2 : \mathsf{num}}{\Gamma \vdash \mathsf{plus}(e_1, e_2) : \mathsf{num}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{str} \quad \Gamma \vdash e_2 : \mathsf{str}}{\Gamma \vdash \mathsf{cat}(e_1, e_2) : \mathsf{str}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}(e_1, x.e_2) : \tau_2}$$

Figure 1: Static semantics for $\mathcal{L}\{$`num`,`str`$\}$.

datatype `exp1`. (In `exp1` the subterms have type `exp`, so you can't see any further without calling `show` again.) In order to create an ABT, you must make an `exp1` (possibly containing some `exp`s you have around) and `hide` it. The key point is that *when you `show` an `exp` representing a `Let` term, an implementation must return a fresh name.* The fact that there is no way to take apart an ABT besides calling `show` means that you can never forget to freshen a bound variable.

Task 1 (10%). Implement a module with signature `EXP` in the file `exp-named.sml` by adapting your code from Homework 1. In your implementation, `exp` and `exp1` should actually be the same type, and `hide` does not need to do anything. `show` should freshen bound names using a function like `swapExp` from Homework 1. Note that your implementation of substitution from Homework 1 does not go in this file; you will use that later.

You can test your code using the function `Top.loop_print` described in the next section. You should be able to see the variables being freshened when it prints a term, because it prints using `show`; for example:

```
- Top.loop_print();

EXP>let x be 4 in x + x end;
let x1517 be 4 in
    x1517 + x1517
end;

EXP>let x be 4 in x + x end;
let x1820 be 4 in
    x1820 + x1820
end;
```

3 Implementing the Static and Dynamic Semantics

In Figure 1, we have included the static semantics of $\mathcal{L}\{$`num`,`str`$\}$. You can find a discussion of this typing judgement in PFPL Chapter 4. Note that we called the types `nat` and `string` rather than `num` and `str` in class. `cat(e1,e2)` is what we wrote as `e1 ^ e2`.

Task 1 (20%). Program a typechecker for $\mathcal{L}\{$`num`,`str`$\}$. We have provided a stub for you in `check.sml`,

$$\frac{\mathsf{add}(m,n,p)}{\mathsf{plus}(\mathsf{num}[m],\mathsf{num}[n]) \mapsto \mathsf{num}[p]} \qquad \frac{\mathsf{append}(s,t,u)}{\mathsf{cat}(\mathsf{str}[s],\mathsf{str}[t]) \mapsto \mathsf{str}[u]}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{plus}(e_1,e_2) \mapsto \mathsf{plus}(e_1',e_2)} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathsf{plus}(e_1,e_2) \mapsto \mathsf{plus}(e_1,e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{cat}(e_1,e_2) \mapsto \mathsf{cat}(e_1',e_2)} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathsf{cat}(e_1,e_2) \mapsto \mathsf{cat}(e_1,e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{let}(e_1,x.e_2) \mapsto \mathsf{let}(e_1',x.e_2)} \qquad \frac{e_1\ \mathsf{value}}{\mathsf{let}(e_1,x.e_2) \mapsto [e_1/x]\ e_2}$$

Figure 2: Dynamic semantics for $\mathcal{L}\{\mathtt{num},\mathtt{str}\}$.

$$\frac{}{[e/x]\ x = e} \qquad \frac{x \neq y}{[e/x]\ y = y}$$

$$\frac{}{[e/x]\ \mathsf{num}[n] = \mathsf{num}[n]} \qquad \frac{}{[e/x]\ \mathsf{str}[s] = \mathsf{str}[s]}$$

$$\frac{[e/x]\ e_1 = e_1' \quad [e/x]\ e_2 = e_2'}{[e/x]\ \mathsf{plus}(e_1,e_2) = \mathsf{plus}(e_1',e_2')} \qquad \frac{[e/x]\ e_1 = e_1' \quad [e/x]\ e_2 = e_2'}{[e/x]\ \mathsf{cat}(e_1,e_2) = \mathsf{cat}(e_1',e_2')}$$

$$\frac{[e/x]\ e_1 = e_1'}{[e/x]\ \mathsf{let}(e_1,x.e_2) = \mathsf{let}(e_1',x.e_2)} \qquad \frac{[e/x]\ e_1 = e_1' \quad x \neq y \quad y \notin e \quad [e/x]\ e_2 = e_2'}{[e/x]\ \mathsf{let}(e_1,y.e_2) = \mathsf{let}(e_1',y.e_2')}$$

Figure 3: Substitution for $\mathcal{L}\{\mathtt{num},\mathtt{str}\}$.

which should result in a structure matching the signature `CHECK` in `check-sig.sml`. We have set things up so that your type checker is a functor parameterized by a structure matching `EXP`. You will need to choose a representation for contexts Γ; don't worry about efficiency for this assignment.

In Figure 2, we have included the dynamic semantics of $\mathcal{L}\{\texttt{num},\texttt{str}\}$. You can find a discussion of this transition system judgement in PFPL Chapter 5. In Figure 3 we have included the definition of substitution on $\mathcal{L}\{\texttt{num},\texttt{str}\}$ terms (this is just a specialization of the definition in PFPL Chapter 1).

Task 2 (20%). Implement the dynamic semantics for $\mathcal{L}\{\texttt{num},\texttt{str}\}$. Once again, we have provided stubs for you in `step.sml`, which should result in a structure matching the signature `STEP` in `step-sig.sml`, and your evaluator should be a functor parameterized by the actual implementation of `EXP`.

To implement these dynamic semantics, you will need to implement substitution. This should be similar to your implementation from Homework 1, except, because you are implementing it from the "outside" of the `EXP` interface, you don't need to explicitly freshen names—`show` does that for you. The freshness condition means that the conditions on the rule for substituting into an abstractor are always satisfied.

In the support code, we have supplied a parser for a concrete syntax for $\mathcal{L}\{\texttt{num},\texttt{str}\}$ as well as several functions to help you test your code (see the `Top` module)

```
    val loop_print : unit -> unit (* just print the same expression back *)
    val loop_type  : unit -> unit (* just type check *)
    val loop_eval  : unit -> unit (* type check and show the final value *)
    val loop_step  : unit -> unit (* type check and show all steps of evaluation *)

    (* similar, but read a .exp source file *)
    val file_print : string -> unit
    val file_type  : string -> unit
    val file_eval  : string -> unit
    val file_step  : string -> unit
```

Some examples are in the file `examples.exp`.

For example, once you're done, you will be able to do this to type check and step through the execution of an expression:

```
- Top.loop_step();
Exp>let x be 4 in x + x end;

let x24 be 4 in
    x24 + x24
end : num;

Press return:

4 + 4 : num;

Press return:

8 : num;
```

You should not change any files other than `exp-named.sml`, `check.sml`, and `step.sml`. We are going to copy these files out of your handin directory and compile them against the original versions of the support files; if they don't compile you will get little credit.

As a general hint, your code should bear an obvious resemblance to the judgements we have discussed. If you find yourself writing a lot of code, or code which you can't explain in terms of the judgements, you are on the wrong track. Don't worry about efficiency; in this assignment we are concerned only with the correctness and clarity of your code.