Concepts of Programmig Language Design

MinHs Exercises

Gabriele Keller

November 6, 2024

1. **Extending MinHs:** Add let-bindings (with a single binding, like in our arithmetic expression language) to MinHs and extend the static semantics accordingly.

   How would the static semantics rule change if we extend the scope of the bound variable to the right hand side of the binding (like in Haskell)?

   Does the following expression type check according to your rules?

   ```
   let
      f = recfun g (Int -> Int) x = if x == 0 then 0 else 1 + f (x-1)
   in f 1
   ```

   If not, is this fixable?

   Solution:

   $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{Let}\ e_1\ (x.e_2)) : \tau_2}$$

   If x can occur in e_1, then the user needs to provide the type for x, otherwise we would have to infer the type, which is more complicated than type checking. Therefore, the concrete syntax could be something like:

   ```
   let
      f : (Int -> Int) = recfun g (Int -> Int) x = if x == 0 then 0 else 1 + f (x-1)
   in f 1
   ```

   And the abstract syntax and inference rule (note that in the abstract syntax representation, it's clear that x is bound in e_1 and e_2):

   $$\frac{\Gamma \cup \{x : \tau_1\} \vdash e_1 : \tau_1 \quad \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{Let}\ \tau_1\ (x.(e_1, e_2))) : \tau_2}$$

2. **Type safety in MinHs :** Give a (partial) proof for type safety in MinHs. For a full proof, we need to show that

   - Progress: If we can derive $\vdash e : \tau$, then either e is a final value, or there is e' such that $e \mapsto e'$.
   - Preservation: If $e \mapsto e'$, and $\vdash e : \tau$ then $\vdash e' : \tau$

   Give for both progress and preservation all the base cases for the proof, and all inductive cases for `if`-expressions.

   Solution: Progress:

   - Base cases: $\vdash e : \tau$ is derivable via an axiom only if e is either a number, boolean value or a function. In all these cases, e is already a final value.

- Inductive case: $e =$ `(If` e_1 e_2 e_3`)` and $\vdash$ `(If` e_1 e_2 e_3`)` : τ. Since there is exactly one typing rule for each form of expression, we can use rule inversion, and conclude the

  A-1 $\vdash e_1$: `Bool`

  By induction hypothesis, we know that either e_1 is final, or there is e_1' with $e_1 \mapsto e_1'$. If it is final, then it is either `True` or `False`, so $e \mapsto e_2$ or $e \mapsto e_3$. If it is not final, then `(If` e_1 e_2 e_3`)` $\mapsto$ `(If` e_1' e_2 e_3`)`, so the evaluation is not stuck.

Preservation:

- Base cases: all the cases where $e \mapsto e'$ via an axiom. For example, addition where both arguments are evaluated, if-expressions where the condition is fully evaluated. In each of these cases, we can use rule inversion to show that the type stays the same.
- Inductive case: $e =$ `(If` e_1 e_2 e_3`)` $\mapsto$ `(If` e_1' e_2 e_3`)`. Again, using rule inversion we know that if $\vdash$ `(If` e_1 e_2 e_3`)` : τ, then

  A-1 $\vdash e_1$: `Bool`

  A-2 $\vdash e_2 : \tau$

  A-3 $\vdash e_3 : \tau$

  Using (A-1) and the induction hypothesis, we have e_1' : `Bool`, and therefore with (A-2) and (A-3), and the typing rule for if-expressions, we can derive $\vdash$ `(If` e_1' e_2 e_3`)` : τ

3. **Equational reasoning in (purely) functional languages.** In Haskell, computations which depend on an explicit state (such as computations of type `IO`) can be composed in a user-friendly way using the `do`-notation. However, this notation can be used for any type constructor `m` for which we can define the function `return :: a -> m a` and the infix operator `(>>=) :: m a -> (a -> m b) -> m b` such that the following equalities hold:

   - `return a >>= f` is equivalent to `f a`
   - `m >>= return` is equivalent to `m`
   - `(m >>= f) >>= g` is equivalent to `m >>= (\x -> f x >>= g)`

   For the type constructor `Maybe` we can define these functions as follows:

```haskell
data Maybe a = Nothing | Just a

return :: a -> Maybe a
return a = Just a

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing  f = Nothing
(>>=) (Just a) f = f a
```

   Show that the equalities listed above (also called Monad laws) indeed hold for any type correct application.

Solution:

1. Show that `return a >>= f` is equal to `f a`

```haskell
  (return a >>= f)
= -- def of return
  (Just a >>= f)
= -- def of (>>=)
  f a
```

2. Show that `m >>= return` is equivalent to `m`

```
-- case m = Nothing
  Nothing >>= return
= -- def of (>>=)
  Nothing


-- case m = Just a
  Just a >>= return
= -- def of (>>=)
  return a
= -- def of (return)
  Just a
```

3. Show that `(m >>= f) >>= g` is equivalent to `m >>= (\x -> f x >>= g)`

```
-- case m = Nothing
  {(Nothing >>= f) >>= g
=  -- def of (>>=)
  Nothing >>= g
=  -- def of (>>=)
  Nothing
=  -- def of (>>=)
  Nothing >>= (\x -> f x >>= g)


-- case m = Just a
  (Just a >>= f) >>= g
=  -- def of (>>=)
   f a >>= g
=  -- application
   (\x -> f x >>= g) a
=  -- def of (>>=)
  Just a >>= (\x -> f x >>= g)
```