```haskell
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}
{-@ LIQUID "--short-names" @-}
{-# LANGUAGE GADTs #-}

module Lec_03_01 where

import ProofCombinators
import Expressions
import Imp
import BigStep
import SmallStep
import qualified State as S
```

1 Introduction

We're continuing small-step semantics today. Recall the form of small-step semantics:

$$(c, s) \leadsto (c', s'),$$

where (c, s) is a "configuration" made up of a command c and a state s. We represent this in Haskell code as `(SStep c s c' s')`.

Today we want to prove two properties about our small-step semantics:

1. Our semantics is *deterministic*: if $(c, s) \leadsto (c_1, s_1)$ and $(c, s) \leadsto (c_2, s_2)$, then $c_1 = c_2$ and $s_1 = s_2$.

$$\begin{tikzcd} & (c_1, s_1) \arrow[d, equal] \\ (c, s) \arrow[ur, squiggly] \arrow[dr, squiggly] & \\ & (c_2, s_2) \end{tikzcd}$$

2. Our small-step semantics is in some sense equivalent to our big-step semantics. Why is this hard?
   - We have to talk about the "final state" in our small-step semantics, which means
   - we have to somehow describe executing the "entire program".

Last time, we started on these goals by proving that $(\texttt{Skip}, s)$ can't step to any other configuration:

```haskell
{-@ lem_not_skip :: c:_ -> c':_ -> s:_ -> s':_ ->
      Prop (SStep c s c' s') -> {c /= Skip} @-}
lem_not_skip :: Com -> Com -> State -> State -> SStep -> Proof
lem_not_skip = undefined
```

2 Our small-step semantics is deterministic

Let's phrase this property in Liquid Haskell:

In class, we wrote this function returning `()`, not `Proof`. This is fine, however, because `Proof` is just a type alias for `()`.

```
{-@ lem_ss_det :: c:_ -> s:_ -> c1:_ -> s1:_ -> c2:_ -> s2:_ ->
      Prop (SStep c s c1 s1) ->
      Prop (SStep c s c2 s2) ->
      {c1 == c2 && s1 == s2} @-}
lem_ss_det :: Com -> State -> Com -> State -> Com -> State ->
      SStep -> SStep -> Proof
```

Now, we can split cases on the different small-step proofs.

In the `SAssign` case:

```
lem_ss_det c s c1 s1 c2 s2 (SAssign {}) (SAssign {})
```

we know that

- `c == Assign x a`,
- `c1 == c2 == Skip`, and
- `s1 == s2 == asgn x a s`.

So we can rely on Liquid Haskell to just figure out the proof for us:

```
  = ()
```

Similarly, in the `SSeq1` case:

```
lem_ss_det c s c1 s1 c2 s2 (SSeq1 {}) (SSeq1 {})
```

we know that

- `c == Seq Skip c'`,
- `c1 == c2 == c'`, and
- `s1 == s2 == s`.

Again this is is enough for Liquid Haskell to figure the proof out:

```
  = ()
```

The `SSeq2` case is more complex:

```
lem_ss_det c s c1 s1 c2 s2
  (SSeq2 cA cA1 cB _s _s1 cA_s_cA1_s1)
  (SSeq2 _  cA2 _  _  _s2 cA_s_cA2_s2)
```

Here, we know

- `c == Seq cA cB`,
- `c1 == Seq cA1 cB`, and
- `c2 == Seq cA2 cB`, as well as

- `cA_s_cA1_s1` (that is, $(cA, s) \rightsquigarrow (cA1, s1)$), and
- `cA_s_cA2_s2` ($(cA, s) \rightsquigarrow (cA2, s2)$).

We need to prove that `c1 == c2 && s1 == s2`, which simplifies to proving `cA1 == cA2 && s1 == s2`. To do this, we can use `lem_ss_det` inductively:

```
  = lem_ss_det cA s cA1 s1 cA2 s2 cA_s_cA1_s1 cA_s_cA2_s2
```

Now for the `SWhileT` case:

```
lem_ss_det c s c1 s1 c2 s2
  (SWhileT b  body  _s)
  (SWhileT _b _body _)
```

We know here that

- `c1 == c2 == Seq body (While b body)` and
- `s1 == s2 == s`.

This is exactly what we need, so Liquid Haskell proves this case automatically:

```
  = ()
```

The `SWhileF`, `SIfT`, and `SIfF` cases are similar and can all be proven automatically:

```
lem_ss_det c s c1 s1 c2 s2 (SWhileF {}) (SWhileF {}) = ()
lem_ss_det c s c1 s1 c2 s2 (SIfT {}) (SIfT {}) = ()
lem_ss_det c s c1 s1 c2 s2 (SIfF {}) (SIfF {}) = ()
```

We've handled all of the constructors for `SStep`. Are we done? Liquid Haskell doesn't think so:

```
Error: Liquid Type Mismatch

 103 | lem_ss_det c s c1 s1 c2 s2 (SAssign {}) (SAssign {})
       ^^^^^^^^^^

   Inferred type
     VV : {v : GHC.Prim.Addr# | v == "function lem_ss_det"}

   not a subtype of Required type
     VV : {VV : GHC.Prim.Addr# | 5 < 4}
```

As we've seen before, this error means we're missing a case. But what case? Recall that we divided the cases where `c == Seq cA cB` into cases where both proofs are either `SSeq1` or `SSeq2`. But what if one is `SSeq1` and the other is `SSeq2`?

```
lem_ss_det c s c1 s1 c2 s2
  (SSeq1 {})
  (SSeq2 cA cA' cB2 _s _s2 cA_s_cA'_s2)
```

Here, the first proof tells us that `c == Seq Skip cB1`, while the second proof tells us that `c == Seq cA cB2`. So we know that `cA == Skip` and `cB1 == cB2`. However, the second proof also tells us that $(cA, s) \rightsquigarrow (cA', s2)$, that is, that $(\texttt{Skip}, s) \rightsquigarrow (\texttt{Skip}, s2)$. We know that this is impossible: we proved that last class. We can thus dismiss this case by using that theorem:

```
  = impossible ("Skip can't step" ? lem_not_skip cA cA' s s2 cA_s_cA'_s2)
```

Finally, we need the opposite case, where the first proof is `SSeq2` and the second is `SSeq1`:

```
lem_ss_det c s c1 s1 c2 s2
  (SSeq2 cA cA' cB1 _s _s1 cA_s_cA'_s1)
  (SSeq1 {})
  = impossible ("Skip can't step" ? lem_not_skip cA cA' s s1 cA_s_cA'_s1)
```

3 Our small- and big-step semantics are equivalent

How can we go from our small-step semantics to "executing whole program"? We want to say that if

$$(c, s) \rightsquigarrow (c_1, s_1) \rightsquigarrow (c_2, s_2) \rightsquigarrow \cdots \rightsquigarrow (\texttt{Skip}, s'),$$

then

$$c : s \Rightarrow s'$$

(and vice-versa).

One idea (from Michael): work backward. First, we prove that if

$$(c_n, s_n) \rightsquigarrow (\texttt{skip}, s'),$$

then

$$c_n : s_n \Rightarrow s'.$$

That is, we prove our semantics is correct when it only needs to take a single step. Once we've proved this, we proceed inductively. We prove that if

$$(c_{n-1}, s_{n-1}) \rightsquigarrow (c_n, s_n),$$

then

$$c_{n-1} : s_{n-1} \Rightarrow s'.$$

Let's try to prove this first lemma. Phrased in Liquid Haskell, we have

```
{-@ lem_michael :: c:_ -> s:_ -> s':_ ->
      Prop (SStep c s Skip s') -> Prop (BStep c s s') @-}
lem_michael :: Com -> State -> State -> SStep -> BStep
```

We case split on the small-step proof. Let's start with the `SAssign` case:

```
lem_michael c s s' (SAssign x a _s)
```

Here we know that

- `c == Assign x a s` and
- `s' == asgn x a s`.

We only have one way to produce a big-step proof for an assignment statement, so we can just use that:

```
  = BAssign x a s
```

We'll look at the rest of the cases on Monday.

```
lem_michael c s s' cs_skips' = undefined
```