Assignment #3:  
System F and PCF  

15-312: Principles of Programming Languages (Spring 2017)  
Out: Wednesday, February 22, 2017  
Due: Saturday, March 11, 2017 11:59pm EST  

Introduction  
In this assignment we study two distinctively different flavors of programming languages. First we investigate System F, a minimal language supporting parametric polymorphism of the kind you’ve encountered in SML. We will see that many of the common types we’ve encountered throughout this course so far can be expressed in this way. Even more, we’ll explore a powerful reasoning technique called “parametricity” for proving things about polymorphic code.  
After that we’ll switch gears entirely and start our investigation of PCF, a small functional language with support for general recursion through a fix point operator.  

Submission  
As usual we’ve provided a makefile in the handout directory. You must run this makefile which will produce a tar file containing your code to be submitted via autolab.  
The check problem you see there will indicate whether or not your code compiled but it isn’t worth any real points on its own. It’s purely for your benefit to check that your submission is well formed.  
You must submit your PDF separately on Gradescope.  

1 Generic Programming with Inductive and Coinductive Types  
Generic programming is key to giving a general account of inductive (finitary) and coinductive (infinitary) types. Traditional “box and pointer” diagrams make sense only for a limited case of inductive types (those with eager constructors), and are completely useless for dealing with laziness, coinductive types, or functions. The generic extension, as we will see, deals with this more general class of inductive and coinductive types nicely.  
The definitions of inductive and coinductive types makes use of generic extension. Having defined these, it is then an interesting exercise to extend generic extension to account for inductive and coinductive types. The interesting part of the problem is to ensure that the generic extension is well-defined. In the case of a polynomial type operator this is ensured by induction on the structure of types. But when extending to inductive and coinductive types the justification is not so straightforward, essentially because it must appeal to the generic extension to the “unrolling” of the inductive or coinductive type, which is larger than the inductive or coinductive type itself.
The trick is to ensure that the generic extension operation satisfies a *regularity* condition stating that it acts as the identity on constant families. Specifically, we must make the following extension to the definition of the generic extension:

\[
\text{map}\{t.\tau\}(x.e')(e) \mapsto e \quad (t \notin \tau)
\]

In other words the generic extension of a function to a constant type family is the identity.

Let \(\tau^u\) be a polynomial type in two variables, \(t\) and \(u\), and write \(\tau^\rho_{\mu} \) for \([\rho/t][\mu/u]\tau\). Supposing that the type of the map to be extended is \(x : \rho \vdash e' : \sigma\), the dynamics of generic extension for inductive and coinductive types are as follows:

\[
\text{map}\{t.\mu_{\rho}(u.\tau^u)\}(x.e')(e) \mapsto \text{rec}\{u.\tau^u\}(y.\text{fold}\{u.\tau^u\}(\text{map}\{t.\mu_{\rho}(u.\tau^u)\}(x.e')(y)); e)
\]

\[
\text{map}\{t.\nu_{\mu}(u.\tau^u)\}(x.e')(e) \mapsto \text{gen}\{u.\tau^u\}(y.\text{map}\{t.\nu_{\rho}(u.\tau^u)\}(x.e')\text{unfold}\{u.\tau^u\}(y)); e)
\]

To get a feel for the dynamics of generic extension for inductive and coinductive types, let’s consider how generic extension could be used to increment all elements of a list containing natural numbers.

Recall that a list containing natural numbers can be defined as

\[
\text{list} \triangleq \mu(1 + (\text{nat} \times u))
\]

and let \(\tau = 1 + (t \times u)\), again writing \(\tau^\rho_{\mu} \) for \([\rho/t][\mu/u]\tau\). Let \(\phi_t\) be the type operator \(u.\tau^u\).

Suppose that \(x.\text{succ}(x)\) is the map to be extended. Suppose that \(e : \mu(\phi_{\text{nat}})\).

**Task 1.1** (5 pts).

Recall that the only rule for the statics of generic extension is

\[
\frac{t.\tau \text{ poly} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\text{map}\{t.\tau\}(x.e')(e) : [\rho'/t]\tau}
\]

Show that the generic extension is well-typed for the inductive case: Given \(e, e' = \text{succ}(x)\) as above, show that \(\text{map}\{t.\mu_{\rho}(u.\tau^u)\}(x.e')(e)\) is well-typed. You do not need to show \(t.\mu_{\rho}(u.\tau^u)\) poly.

**Task 1.2** (5 pts). Evaluate

\[
\text{map}\{t.\mu_{\rho}(u.\tau^u)\}(x.\text{succ}(x))\text{fold}\{u.1 + \text{nat} \times u\}(r \cdot (\text{succ}(\text{zero}), \text{fold}\{u.1 + \text{nat} \times u\}(1 \cdot ())))
\]

showing each step.

Let’s define infinite streams of naturals as \(\text{infstream} \triangleq \nu(\text{nat} \times u)\).

Define

\[
\text{zeros} = \text{gen}\{u.\text{nat} \times u\}(x.(x,x); \text{zero})
\]

\[
\text{head} = \lambda(\text{infstream} : s) \text{unfold}\{u.\text{nat} \times u\}(s \cdot 1)
\]

**Task 1.3** (5 pts). Evaluate \(\text{head}(\text{zeros})\). You can find the relevant dynamics rules for map in the appendix of this assignment.
Task 1.4 (5 pts). Below, we provide several intermediate steps in the evaluation of

\[
\text{head}(\text{map}\{t.\nu(u.t \times u)\}(x.\text{succ})(\text{zeros}))
\]

. Perform a single step at each intermediate step labelled with a question mark.

\[
\text{head}(\text{map}\{t.\nu(u.t \times u)\}(x.\text{succ})(\text{zeros}))
\]

\[
\rightarrow \text{unfold}_{u.\text{nat} \times u}(\text{gen}_{u.\text{nat} \times u}(\text{y.map}\{t.t \times \nu(u.t \times u)\}(x.\text{succ})(\text{unfold}_{u.\text{nat} \times u}(y)); \text{zeros}) \cdot l
\]

\[
\rightarrow ?
\]

\[
\rightarrow \text{map}\{u.\text{nat} \times u\}(\text{z.gen}_{u.\text{nat} \times u}(\text{y.map}\{t.t \times \nu(u.t \times u)\}(x.\text{succ})(\text{unfold}_{u.\text{nat} \times u}(y)); \text{z})

(\text{map}\{t.t \times \nu(u.t \times u)\}(x.\text{succ})(\text{unfold}_{u.\text{nat} \times u}(\text{gen}_{u.\text{nat} \times u}(x.(x,x); \text{zero}))) \cdot l
\]

\[
\rightarrow ?
\]

\[
\rightarrow \langle \text{map}\{t.t\}(x.\text{succ})(\text{zero}), \text{map}\{t.\nu(t \times u)\}(x.\text{succ})(\text{gen}_{u.\text{nat} \times u}(x.(x,x); \text{zero})) \cdot l
\]

\[
\rightarrow ?
\]

2 System FE

In the last 10 or so years a feature called “generics” or “parametric polymorphism” has swept through the programming languages that see common use. It’s usually sold as a way to avoid duplication of code. For example, thanks to polymorphism in SML, we can have a single \text{map} function which works for all lists, no matter the items stored in the list. Clearly this is a useful feature and so we can turn the tools that we’ve used in 15-312 so far to investigate exactly what the heart of parametric polymorphism is. The language we use to study this is System F (which has been around since the 1960s!).

System F contains only a few constructs. Some should be familiar, including functions and application but there are two new constructs that are designed to isolate the core idea of polymorphism. We can abstract over a type in a term and later on specialize it. This is almost identical to the process of abstracting over a term like we do in a function. The syntax of the language follows:
<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract form</th>
<th>Concrete form</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>$\tau ::= t$</td>
<td>$t$</td>
</tr>
<tr>
<td></td>
<td>$\text{arr}(\tau_1;\tau_2)$</td>
<td>$\tau_1 \to \tau_2$</td>
</tr>
<tr>
<td></td>
<td>$\text{all}(t.\tau)$</td>
<td>$\forall(t.\tau)$</td>
</tr>
<tr>
<td></td>
<td>$\text{some}(t.\tau)$</td>
<td>$\exists(t.\tau)$</td>
</tr>
<tr>
<td></td>
<td>nat</td>
<td>nat</td>
</tr>
</tbody>
</table>

| exp    | $e ::= x$      | $x$           |
|        | $\text{lam}\{\tau\}(x.e)$ | $\lambda (x : \tau) e$ |
|        | $\text{ap}(e_1;e_2)$    | $e_1(e_2)$ |
|        | $\text{Lam}(x.e)$       | $\Lambda(x)e$ |
|        | $\text{App}\{\tau\}(e)$ | $e[\tau] $ |
|        | zero             | zero          |
|        | $\text{succ}(e)$    | $\text{succ}(e)$ |
|        | $\text{rec}\{e_2;x.y.e_3\}\{e_1\}$ | $\text{rec} e_1 \{ z \leftarrow e_2 \mid s(x) \text{ with } y \leftarrow e_3 \}$ |
|        | $\text{pack}\{t.\tau\}\{\rho\}\{e\}$ | $\text{pack } \rho \text{ with } e \text{ as } \exists(t.\tau)$ |
|        | $\text{open}\{t.\tau\}\{\rho\}\{e_1;t,x.e_2\}$ | $\text{open } e_1 \text{ as } t \text{ with } x : \tau \text{ in } e_2$ |

There’s one new type in addition to function types which classifies expressions of polymorphic type. The statics of this language are a little more complicated than we’re used to because we also need to include an explicit judgment for ensuring that the types we write down are well formed. Now that we have type variables there’s a possibility of malformed types. This is an issue that will undoubtedly factor into your implementation of this language later on. For simplicity, we’ve also supplemented our language with naturals and existentials because it will make your test programs easier to write. Both constructs need not be primitive though as we shall see.
The important things to notice here are the similarities between the set of constructs for type abstraction (polymorphism) and value abstraction (functions). Additionally, note that we choose to keep two separate contexts, one for type and one for value variables. Next we have the dynamics.
Task 2.1 (40 pts). Now as should be normal by this point, your first pair of tasks is to implement the statics and dynamics of System F. For this you will be modifying `sec2/typechecker.sml` and `sec2/dynamics.sml`. We have provided you with an Abbot based ABT structure for System F as we did in the prior homework so be sure to check how we’ve represented System F programs.

### 2.1 Definability

One of the most interesting aspects of polymorphism is the fact that this one type can be used to implement versions of all the types we’ve discussed so far. For example, pairs and sums may both be represented in pure System F. For pairs, the idea is that we define an appropriate polymorphic type so that any program which belongs to this type may answer queries about what the first and second component of the underlying tuple is. We define

\[ \tau_1 \times \tau_2 \triangleq \forall (u. (\tau_1 \rightarrow \tau_2 \rightarrow u) \rightarrow u) \]
The idea being that once a program has obtained something of type $\tau_1 \times \tau_2$, it may query it with “questions” of the form $\tau_1 \rightarrow \tau_2 \rightarrow u$ and receive answers of type $u$ for any $u$. This allows us to define both projections easily. Projecting out the left component is done with

$$e.l \triangleq e[\tau_1](\lambda (x : \tau_1) \lambda (y : \tau_2) x)$$

And projecting out the right is done with

$$e.r \triangleq e[\tau_2](\lambda (x : \tau_1) \lambda (y : \tau_2) y)$$

Of course, we need to be able to actually construct these encoded pairs. We can encode $\langle e_1, e_2 \rangle$ into this schema in such a way that it coheres with our definition of the projections.

$$\langle e_1, e_2 \rangle \triangleq \Lambda(u)(\lambda (f) f(e_1)(e_2))$$

The general scheme for encoding a type into System $F$ is

1. Start with $\forall (u. \ldots \rightarrow u)$, as all of these are “question and answer” sort of encodings
2. Each different way of forming the original type will be an argument in the ...s
3. If one way of forming something of the original type needs arguments of type $\tau_1$ through $\tau_n$, then it is encoded as $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow u$. If any $\tau_1$ is actually the type we’re encoding (an inductive reference to it) then we replace it with $u$ instead.

We’ve already seen what running this procedure does with products. With sums we can follow the same steps; to encode $\tau_1 + \tau_2$ into pure System $F$ we do the following

1. We start with $\forall (u. \ldots \rightarrow u)$
2. There are two ways of forming sums, left and right injection. Therefore we’ll have two arguments $\forall (u. (\ldots) \rightarrow (\ldots) \rightarrow u)$
3. The first way is left injection and so the first $\ldots$ becomes $\tau_1 \rightarrow u$ and the second becomes $\tau_2 \rightarrow u$. This leaves us with $\forall (u. (\tau_1 \rightarrow u) \rightarrow (\tau_2 \rightarrow u) \rightarrow u)$

We can encode more sophisticated types into pure System $F$. For example, we can encode the existential types you’ve encountered in the implementation. We designed them as primitive to our language for simple convenience but there’s no reason that they have to be!

For this, we follow the same basic recipe but make on the arguments in the recipe above a polymorphic function instead. That is,

$$\exists(t.\tau) \triangleq \forall(u.(\forall(t.\tau \rightarrow u)) \rightarrow u)$$

Notice the similarity between this and how we’ve encoded pairs. Just like before we’re taking all the data we supplied to the pack (respectively $\langle , \rangle$) operator as arguments to a function. Unlike before however now some of that data is a type and so we need to use a polymorphic function.

As before, define the proper System $F$ encodings for pack and open. As a hint: your answer should be almost identical to how we encoded pairs and their eliminators but unlike before we’ll have $\Lambda(\cdot)$ instead of $\lambda (\cdot)$ in a few places.
Since you need type information in your translation, you should assume that your translation is part of an inductively defined judgement

\[ \Gamma \vdash e : \tau \leadsto e' \]

that eliminates `pack` and `open` from a typed FE expression `e` such that \( \Gamma \vdash e : \tau \). For example, the translation rule for applications `\text{ap}(e_1; e_2)` is

\[
\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \leadsto e_1' \quad \Gamma \vdash e_2 : \tau_2 \leadsto e_2'}{\Gamma \vdash \text{ap}(e_1; e_2) : \tau \leadsto \text{ap}(e_1'; e_2')}\]

Task 2.2 (10 pts). Define the translation rules for `\text{pack}\{t.\tau\}\{\rho\}(e)` and `\text{open}\{t.\tau\}\{\rho\}(e_1; t, x, e_2)`.

Consider an extension of System F with products and streams of natural numbers. This extension involves a new type `\text{stream}` along with two operations on it for constructing streams:

\[
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau \vdash e_2 : \text{nat} \quad \Gamma, x : \tau \vdash e_3 : \tau}{\Gamma \vdash \text{streamgen}\{\tau\}(e_1; x. e_2; x. e_3) : \text{stream}} \quad \Gamma \vdash e : \text{stream} \quad \Gamma \vdash \text{hd}(e) : \text{nat}
\]

\[
\frac{\Gamma \vdash e : \text{stream}}{\Gamma \vdash \text{tl}(e) : \text{stream}} \quad \frac{\Gamma \vdash \text{streamgen}\{\tau\}(e_1; x. e_2; x. e_3) \text{ val}}{e \mapsto e'} \quad \frac{e \mapsto e'}{\text{hd}(e) \mapsto \text{hd}(e')} \quad \frac{\text{tl}(e) \mapsto \text{tl}(e')}{e \mapsto e'} \quad \frac{\text{hd}(\text{streamgen}\{\tau\}(e_1; x. e_2; x. e_3)) \mapsto [e_1/x]e_2}{\text{streamgen}\{\tau\}(e_1; x. e_2; x. e_3) \mapsto \text{streamgen}\{\tau\}([e_1/x] e_3; x. e_2; x. e_3)}
\]

The key idea is that streams are only generated when demanded by a `hd()` or `tl()`. Inside, streams have some internal state (here called `e_1`) as well as two operations which use the internal state to calculate the next natural number in the sequence or the next internal state.

Task 2.3 (20 pts).

With existentials and products embedded in System F, we don’t need to extend the language at all to get streams!

Encode the type of streams outlined above in pure System F using existentials. Then translate the core operations: head, tail, and `\text{genstream}`. You can test your translation by running the examples you suggested above in your System F interpreter after encoding products and streams appropriately.

### 2.2 Representation Independence

One of the most important concepts in all of computer science is abstraction, the ability to define multiple implementations of the same interface and swap them out at will is the cornerstone of large scale software. Languages like SML provide for abstraction with a sophisticated module system allowing for a user to define types so that the implementation details are hidden but a privileged few functions may still manipulate them. While we may not yet have the tools in this class to study SML’s modules, we can certainly study abstraction
through existential types which allow us some of the utility we got out of modules. In this section we’ll prove that two existential “packages” behave the same using a technique called parametricity.

The basic idea of parametricity is that we’ll establish some sort of relation $\mathcal{R}$ between the hidden types of two implementations of some existential package. We then show that this relation is always “preserved” by the operations of the package. Consider this package

$$\text{finite_list} = \exists t. t \times (\text{nat} \to t \to t) \times (t \to \text{nat})$$

The idea is that an implementation of a finite list of naturals is really an existential package of some type representing the list along with a product of 3 operations. The first one is a $t$ which constructs an empty list. The second one adds a nat onto the beginning of an existing list and returns a new list. The third operation returns the head of the list. Ideally the last operation could indicate failure with an exception or option type but to keep matters simple we’ve opted to just return 0 in the case of the third operation failing.

To show that two implementations of this package are equivalent we establish a simulation relationship between $\tau_1$ and $\tau_2$ (the types of the two packages under the hood) and show that it exhibits a few properties that show that it is preserved by the operations of abstract type.

**Task 2.4** (10 pts). In recitation we went through several examples of what these proof obligations look like for other abstract data types. Write down what properties a relation $\mathcal{R}$ must satisfy in order to show that two implementations of $\text{finite_list}$ are equivalent must satisfy.

(Hint: there should be 3 obligations, one for each operation)

For this next question you’ll prove that two implementations of finite lists are equivalent using this scheme. We’ve defined two implementations of this package below.

```plaintext
fn_pack = pack \text{nat} \to \text{nat} with
  \langle fn(n : \text{nat}) z, fn(x : \text{nat}) fn(l : \text{nat} \to \text{nat}) fn(n : \text{nat}) \text{rec}(n; x; \_._n'. l(n')) , fn(l : \text{nat} \to \text{nat}) l(z) \rangle as finite_list

(* For this representation, we need the definition of nil and cons
 * a few times
 *)

nil = tfn(c) fn(nil : c) fn(cons : \text{nat} \to c \to c) nil
cons = fn(n : \text{nat}) fn(l : \forall c. c \to (\text{nat} \to c \to c) \to c)
      tfn(c) fn(nil : c) fn(cons : \text{nat} \to c \to c)
      cons(n) (l[c](nil)(cons))

list_pack = pack \forall c. c \to (\text{nat} \to c \to c) \to c) with
  \langle \text{nil},
    \text{cons},
    fn(l : \forall c. c \to (\text{nat} \to c \to c) \to c)
    \text{[nat]}(z)(fn(h : \text{nat}) fn(next : \text{nat}) h) \rangle
  as finite_list
```

The second implementation warrants some explanation. It’s the System F encoding of finite lists of natural numbers in terms of their recursor. That is, before we had a two ways of forming lists, nil and cons
and a way of recursing based on the fact that lists were the smallest type closed under those operations. This gives us a definition in System F exactly with the procedure we defined above. The nil and cons constants we defined above are just those we had before except retrofitted for this representation.

Task 2.5 (20 pts). Define a relationship $R$ in the style of the above that shows that $\text{fun\_pack}$ and $\text{list\_pack}$ are equivalent. Argue informally why your chosen relation does indeed relate the two packages but we do not expect a formal proof for this. A few sentences in English for each of the 3 operations is sufficient.

Task 2.6 (20 pts). The abstract data type of lists above is somewhat lacking because it doesn’t provide a way to take the tail of a list. Write down a new existential type which duplicates all the operations provided by $\text{finite\_list}$ but also includes a tail operation. Then rewrite $\text{list\_pack}$ so that it satisfies this extended signature (define $\text{tail}$).

(Hint: It’s surprisingly difficult to write tail but it will end up being very similar to a $\text{pred}$ operation on Church numerals, that’s a good place to start. In the end you’ll end up wanting to compute a pair of the tail of the list along with something else and then post processing the result to throw away the extra component of the tuple. The solution in the end though is only slightly longer than an implementation of $\text{head}$ so if you find yourself writing several supporting functions you’ve gone wrong.)

(Additional hint: The key idea of $\text{pred}$ is that $\text{pred}(x)$ counts up to $x$, starting from 1 instead of 0.)

3 PCF

We now shift our attention away from System F and polymorphism onto a very different language: PCF, the language of partial, computable functions. PCF is very similar to the language you implemented previously (Gödel’s T) but instead of opting for a recursor on the natural numbers PCF provides only an expression to case on whether or not a number is zero. In order to make up for the lost expressive power (we have no iteration now!) it provides an entirely different operator called $\text{fix}$. This introduces general recursion into the language making it possible for us to write some interesting programs in pure PCF.

The syntax of the language is

<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract form</th>
<th>Concrete form</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>$\tau ::= \text{nat}$</td>
<td>$\text{nat}$</td>
</tr>
<tr>
<td></td>
<td>$\text{arr}(\tau_1;\tau_2)$</td>
<td>$\tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td>exp $e ::= x$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td>$\text{lam}{\tau}(x.e)$</td>
<td>$\lambda (x:\tau) e$</td>
</tr>
<tr>
<td></td>
<td>$\text{ap}(e_1;e_2)$</td>
<td>$e_1(e_2)$</td>
</tr>
<tr>
<td></td>
<td>$\text{zero}$</td>
<td>$\text{zero}$</td>
</tr>
<tr>
<td></td>
<td>$\text{succ}(e)$</td>
<td>$\text{succ}(e)$</td>
</tr>
<tr>
<td></td>
<td>$\text{ifz}{e_2; x.e_3}(e_1)$</td>
<td>$\text{ifz} e_1 {z \rightarrow e_2 \mid s(x) \rightarrow e_3}$</td>
</tr>
<tr>
<td></td>
<td>$\text{fix}{x}{\tau.e}$</td>
<td>$\text{fix}\tau : x \text{ is } e$</td>
</tr>
</tbody>
</table>

The statics and the dynamics of the language are very similar to what we’ve seen, the only surprising rules being those concerning the new operator for $\text{fix}$. 

10
1. Define the shortest program possible which loops forever when evaluated. This program should not be stuck, that is, it should be the case that \( e \mapsto e' \) holds for some \( e' \) for your chosen \( e \).

2. Write a function which subtracts one number from another. If the result should be negative then return \( 0 \).

3. Implement integer division. Your implementation should round up so that \( \text{div}(2)(3) \) is 1 not 0.

4. Implement \( \text{mod} \), it should be the case that \( \text{div}(n)(m) \ast m + \text{mod}(n)(m) = n \) once you’re done assuming that you defined \( \ast \) appropriately.

Note that for this problem your function may behave however you choose for inappropriate inputs. Specifically, you may do whatever is easiest when you encounter division by zero or something similar, the reference solution loops.
Appendix 1: Testing Implementations

You are expected to test your implementations of System F and PCF. We’ve provided a general framework for defining and running tests in both of these languages but they are woefully incomplete. We’ve provided you with 3 main tools for testing:

1. A reference solution for each interpreter is provided. You can run `sml @SMLload refsolN.x86-linux` and SML/NJ will spin up a REPL loaded with our implementation of the language. From there you may experiment to see how your code should behave for some input.

2. A REPL is available using `TopLevel.repl ()`. The set of commands available are identical to the last assignment, you should use this to experiment with small programs.

3. And as before, you may write small test cases in `tests.sml` and run them using `TestHarness.runalltests`.

Appendix 2: Selected Dynamics of Generic Extension and Coinductive Types

\[
\begin{align*}
\text{unfold}\{t.\tau\}(\text{gen}\{t.\tau\}(x.e_1; e_2)) & \mapsto \text{map}^+\{t.\tau\}(y.\text{gen}\{t.\tau\}(x.e_1; y))([e_2/x]e_1) \\
\text{map}\{t.\tau_1 \times \tau_2\}(x.e')(e) & \mapsto \langle \text{map}\{t.\tau_1\}(x.e')(e \cdot l), \text{map}\{t.\tau_2\}(x.e')(e \cdot r) \rangle \\
\text{map}\{t.t\}(x.e')(e) & \mapsto [e/x]e'
\end{align*}
\]