

# Type Systems for Programming Languages (15-814)

## Lecture Notes, Fall 2006, Week 5

Xinyu Zhuang

October 10 and 12, 2006

### 1 Previously...

In the previous lecture, we showed how *Tait's Method* can be used to prove termination of Gödel's T. The crux of the argument lay in our strengthening of the inductive hypothesis by defining Hereditary Termination ( $\text{HT}_\tau(e)$ ).

*Tait's Method* is also known as the *Method of Logical Relations*. Types induce a relational action; for example, in:

$$\text{HT}_{\tau_1 \rightarrow \tau_2} = \text{HT}_{\tau_1} \rightarrow \text{HT}_{\tau_2}$$

the type  $\tau_1 \rightarrow \tau_2$  induces the action  $\rightarrow$  on predicates  $\text{HT}_{\tau_1}$  and  $\text{HT}_{\tau_2}$ .

The direct consequence of our proof of termination is that the type  $\tau_1 \rightarrow \tau_2$  in Gödel's T consists of only total (terminating) functions;  $\text{nat} \rightarrow \text{nat}$  is a space of number-theoretic functions.

The proof of termination of any function in Gödel's T is thus embedded within the function definition itself. This is why some seemingly simple programs have complicated definitions in Gödel's T: the code has to show how the function will always terminate.

### 2 PCF

PCF is short for *Programming Language for Computable Functions*. It makes two key changes to Gödel's T:

1. Replace primitive recursion with case analysis:

$$\text{natcase}(e; e_0; x.e_1)$$

2. General recursion:

$$\text{fix}(x.e)$$

#### 2.1 Static Semantics

$$\frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat} \vdash e_1 : \tau}{\Gamma \vdash \text{natcase}(e; e_0; x.e_1) : \tau} \text{CASE} \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix}(x : \tau.e) : \tau} \text{FIX}$$

The rule CASE is similar to the ML case construct, but only for nats. Also note the many occurrences of type  $\tau$  in the rule FIX: they are all the same type because the variable  $x$  is really a binding for the FIX expression itself.

## 2.2 Dynamic Semantics

$$\frac{}{\underline{\text{natcase}}(\text{zero}; e_0; x.e_1) \mapsto e_0} \qquad \frac{}{\underline{\text{natcase}}(\text{succ}(e); e_0; x.e_1) \mapsto [e/x] e_1}$$

$$\frac{e \mapsto e'}{\underline{\text{natcase}}(e; e_0; x.e_1) \mapsto \underline{\text{natcase}}(e'; e_0; x.e_1)}$$

$$\frac{}{\underline{\text{fix}}(x.e) \mapsto [\underline{\text{fix}}(x.e)/x] e}$$

$\underline{\text{fix}}$  is isomorphic to the Y-combinator. We can define functions such as the factorial function using it:

**Example 2.1.**

$$\overline{\text{fact}} := \underline{\text{fix}}(f : \text{nat} \rightarrow \text{nat} . \lambda x : \text{nat} . \underline{\text{natcase}}(x; \text{succ}(\text{zero}); y . \underline{\text{times}}(\text{succ}(y), f y)))$$

Evaluating  $\overline{\text{fact}}$  gives us:

$$\begin{aligned} \overline{\text{fact}} e &\mapsto ([\overline{\text{fact}}/f] \lambda x . \underline{\text{natcase}}(x; \dots f y \dots)) e \\ &= (\lambda x . \underline{\text{natcase}}(x; \dots \overline{\text{fact}} y \dots)) e \\ &\mapsto \underline{\text{natcase}}(e; \dots \overline{\text{fact}} y \dots) \end{aligned}$$

The progress and preservation theorems of PCF are easily provable.

## 2.3 Termination

Consider the following expression:

$$\Omega_\tau := \underline{\text{fix}}(x : \tau.x)$$

Evaluating this gives us:

$$\begin{aligned} \Omega_\tau &= \underline{\text{fix}}(x : \tau.x) \\ &\mapsto [\underline{\text{fix}}(x : \tau.x)/x] x \\ &\mapsto \underline{\text{fix}}(x : \tau.x) \\ &= \Omega_\tau \end{aligned}$$

$\Omega_\tau$  is thus a diverging expression. So, by adding general recursion into the language, we sacrifice termination;  $\tau_1 \rightarrow \tau_2$  is now a space of *partial* functions<sup>1</sup>.

## 3 Products

*Products* can be thought of as ordered tuples of values. The *nullary product*, often referred to as  $\text{unit}^2$ , is a product of *no* types, and the *binary product* is basically a 2-tuple or pair. The unit type is written as  $\mathbf{1}$ , and the binary product type is written as  $\tau_1 \times \tau_2$ .

<sup>1</sup>The symbol  $\rightarrow$  is sometimes used to refer to partial functions:  $\tau_1 \rightarrow \tau_2$

<sup>2</sup>This is often mistakenly called “void” in languages such as C; the “void” type in C is really the unit type. The real “void” type is discussed in the subsection about sums.

### 3.1 Static Semantics

The unit type has a single intro form and no elim form.

$$\frac{}{\Gamma \vdash \langle \rangle : 1} \text{1-I}$$

We can think of the unit type as representing the “trivial” value. It is often used as the return value of a function where the result is not of interest, most likely because the function has some side-effect which is its main purpose. This is expressed in the static semantics: a value of unit type can be created “out of nothing” (introduction), and there is nothing you can do with a value of a unit type (no elimination).

The product type has the following intro and elim forms:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \times\text{-I}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \underline{\text{fst}} e : \tau_1} \times\text{-E-L} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \underline{\text{snd}} e : \tau_2} \times\text{-E-R}$$

The elim forms fst and snd return the first and second values contained within the pair respectively.

### 3.2 Dynamic Semantics

$$\frac{\langle e_1, e_2 \rangle \text{ value}}{\underline{\text{fst}} \langle e_1, e_2 \rangle \mapsto e_1} \qquad \frac{\langle e_1, e_2 \rangle \text{ value}}{\underline{\text{snd}} \langle e_1, e_2 \rangle \mapsto e_2}$$

$$\frac{e \mapsto e'}{\underline{\text{fst}} e \mapsto \underline{\text{fst}} e'} \qquad \frac{e \mapsto e'}{\underline{\text{snd}} e \mapsto \underline{\text{snd}} e'}$$

We also need value judgements to be able to satisfy the antecedents of the reduction for fst and snd. There are two choices for this: we can have *eager pairs*, or *lazy pairs*.

- Eager Pairs:

$$\frac{e_1 \text{ value} \quad e_2 \text{ value}}{\langle e_1, e_2 \rangle \text{ value}}$$

- Lazy Pairs:

$$\frac{}{\langle e_1, e_2 \rangle \text{ value}}$$

The difference between the two is that the components of an eager pair are always values, and thus have been fully evaluated. The components of a lazy pair have not, and depending on the use of the pair, may never be.

Since we require evaluation of the components of an eager pair, if we choose to use eager pairing we must add a few new rules to the dynamic semantics:

$$\frac{e_1 \mapsto e'_1}{\langle e_1, e_2 \rangle \mapsto \langle e'_1, e_2 \rangle} \qquad \frac{e_1 \text{ value} \quad e_2 \mapsto e'_2}{\langle e_1, e_2 \rangle \mapsto \langle e_1, e'_2 \rangle}$$

Here we evaluate in a left-to-right order, but that a purely arbitrary decision.

Lazy pairs allow for interesting constructions that are not possible using eager pairs<sup>3</sup>. Consider the following example:

**Example 3.1.**

$$\underline{\text{fix}}(x.\langle 3, x \rangle) \mapsto \langle 3, \underline{\text{fix}}(x.\langle 3, x \rangle) \rangle$$

When using lazy pairs, the following evaluation terminates because  $\langle 3, \underline{\text{fix}}(x.\langle 3, x \rangle) \rangle$  is a value<sup>4</sup>. If we were using eager pairs, the evaluation would continue as follows:

$$\begin{aligned} \langle 3, \underline{\text{fix}}(x.\langle 3, x \rangle) \rangle &\mapsto \langle 3, \langle 3, \underline{\text{fix}}(x.\langle 3, x \rangle) \rangle \rangle \\ &\mapsto \langle 3, \langle 3, \langle \dots \rangle \rangle \rangle \\ &\dots \end{aligned}$$

which of course results in divergence and non-termination. In contrast the lazy pair does not, allowing us to construct infinite sequences of 3's. We can extract the left value using  $\underline{\text{fst}}(\langle 3, \underline{\text{fix}}(x.\langle 3, x \rangle) \rangle) \mapsto 3$ , and by using  $\underline{\text{snd}}(\langle 3, \underline{\text{fix}}(x.\langle 3, x \rangle) \rangle) \mapsto \underline{\text{fix}}(x.\langle 3, x \rangle)$ , we get back the “tail” of the infinite sequence. This is similar to ML's streams.

## 4 Sums

The *sum* type is often described as an n-way choice between different types. The *nullary sum* is known as the void type, written as 0. The *binary sum* is written as  $\tau_1 + \tau_2$ , and represents a choice between the two types  $\tau_1$  and  $\tau_2$ . An object of such a type could be of either of the two types.

### 4.1 Static Semantics

Since the nullary sum is a choice between no alternatives, there is no such value to choose from and so it has no intro form. The elim form is as follows:

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \underline{\text{abort}}(e) : \tau} \text{0-E}$$

Note that  $\tau$  can have any type. The meaning of abort is explained later.

Binary sums have the following intro and elim forms:

$$\begin{aligned} \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \underline{\text{inl}}_{\tau_2}(e) : \tau_1 + \tau_2} \text{+I-L} & \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \underline{\text{inr}}_{\tau_1}(e) : \tau_1 + \tau_2} \text{+I-R} \\ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, x : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \underline{\text{case}}(e; x.e_1; x.e_2) : \tau} \text{+E} \end{aligned}$$

In the intro forms, we annotate inl and inr with the *non-injected* type for typechecking purposes. Depending on how we choose to perform typechecking, we may require this additional knowledge in order to proceed.

For example, in type synthesis, we compute the type as a function of the context  $\Gamma$  and the expression. Given an expression inl( $e$ ), we can determine the type of  $e$ , but we have no way of telling what the other half of the sum type is. Thus we require annotation so we can successfully determine the type of inl( $e$ ).

<sup>3</sup>Not entirely true, because you can create lazy pairs out of eager pairs and functions.

<sup>4</sup>Note however that the expression does not typecheck given the rules we have established so far. For that, we need recursive types, which are described later.

An alternative typechecking method known as type analysis does not require annotation, because when typechecking an `inl` or `inr`, it already has the sum type available for checking.

To ease readability we will refrain from adding the type annotations in future uses of `inl` and `inr`.

The `case` construct performs a case analysis on the binary sum  $e$ . Depending on whether  $e$  was constructed using `inl` or `inr`, we branch to either  $e_1$  or  $e_2$ . It is important to note here that whichever branch we choose, we must ensure that both branches yield a result of the same type  $\tau$ . In other words, `case` allows us to return a value of a certain type *no matter what type of expression we are given*, as long as it is one of the values making up the sum type.

## 4.2 Dynamic Semantics

$$\frac{\text{inl}(e) \text{ value}}{\text{case}(\text{inl}(e), x.e_1, x.e_2) \mapsto [e/x] e_1} \qquad \frac{\text{inr}(e) \text{ value}}{\text{case}(\text{inr}(e), x.e_1, x.e_2) \mapsto [e/x] e_2}$$

$$\frac{e \mapsto e'}{\text{case}(e; x.e_1, x.e_2) \mapsto \text{case}(e'; x.e_1, x.e_2)}$$

As can be seen from the dynamic semantics, we branch to  $e_1$  with the expression  $e$  if we case analyse over `inl`( $e$ ), and similar for  $e_2$ .

Just like with pairs, there can be eager sums and lazy sums, depending on our choice of value judgements.

- Eager Sums:

$$\frac{e \text{ value}}{\text{inl}(e) \text{ value}} \qquad \frac{e \text{ value}}{\text{inr}(e) \text{ value}}$$

- Lazy Sums:

$$\overline{\text{inl}(e) \text{ value}} \qquad \overline{\text{inr}(e) \text{ value}}$$

Eager sums require us to add the following judgements to the dynamic semantics:

$$\frac{e \mapsto e'}{\text{inl}(e) \mapsto \text{inl}(e')} \qquad \frac{e \mapsto e'}{\text{inr}(e) \mapsto \text{inr}(e')}$$

## 4.3 The Meaning of `abort`

Consider the function  $f : \text{nat} \rightarrow 0$ . Since there are no intro forms for the void type, this means that there is no way that  $f$  could ever return a value. Thus,  $f$  must be divergent and never return. Another way of thinking about this is that  $f$  never returns to its call site; it is similar to a `goto` statement, or the raising of an exception.

The `abort` construct can be thought of as a case analysis over no cases. This means that it has nothing to branch to, and thus aborts the computation. Thus, the type of `abort` can be anything, since it represents a failure condition.

## 4.4 An Alternative Treatment of Sums

Suppose we decide to use a different set of constructs for the elimination of sums:

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \underline{\text{outl}}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \underline{\text{outr}}(e) : \tau_2}$$

$$\frac{}{\underline{\text{outl}}(\underline{\text{inl}}(e)) \mapsto e} \qquad \frac{}{\underline{\text{outr}}(\underline{\text{inr}}(e)) \mapsto e}$$

This may seem reasonable, but notice that the dynamic semantics are in fact incomplete. Consider the following cases:

$$\underline{\text{outl}}(\underline{\text{inr}}(e)) \mapsto ?$$

$$\underline{\text{outr}}(\underline{\text{inl}}(e)) \mapsto ?$$

Obviously, the two expressions given above do not make sense, yet they typecheck. There is no way of continuing evaluation, but the expressions are not values. The addition of outl and outr has broken type safety in our language.

outl and outr can be thought of as analogous to downcasting in a language such as Java. Given classes (types)  $C_1$  and  $C_2$ , the superclass of both of them is  $C_1 + C_2$ . The Java expression:

$(C_1) \text{ o};$

where  $\text{o}$  is inr( $\text{o2}$ ) (and  $\text{o2}$  has type  $C_2$ ) is the same as outl(inr( $\text{o2}$ )).

We can also think of a pair of constructs analogous to Java's `instanceof` operator: isl and isr.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \underline{\text{isl}}(e) : \text{bool}} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \underline{\text{isr}}(e) : \text{bool}}$$

Here `bool` is the boolean type, which can be expressed as  $1 + 1$ , where inl( $\langle \rangle$ ) is true and inr( $\langle \rangle$ ) is false.

It is interesting to note that in this situation, the only way to make use of isl and isr is within an if-then-else conditional branch, which (since `bool` is defined as a sum type) is really just a case analysis!

However, using isl and isr with an if-then-else statement is actually giving us less power than if we did a case analysis directly on the sum types. Consider this example:

case( $e, x.e_1, x.e_2$ )

In a case analysis, each branch is given the value that was injected into  $e$  as a new binding  $x$ . Thus, the expressions within the branch can make use of the knowledge that  $e$  is really an inl or an inr. With an if-then-else statement, such knowledge is not propagated, which means it is possible that some expression within the branch will have to needlessly test on  $e$  again.

A situation where this is particularly troublesome is in Java, where a value of a certain class can either be an object of that class, or `null`. In other words, we have a sum type of the “null type” and the class. If Java had case analysis and sum types, then it would become unnecessary for functions to have to test whether a provided argument is null or not all the time. ML's polymorphic option type provides precisely this capability, of allowing a value to either be something, or nothing.

## 5 Records and Objects

A *record* is a labelled product. We label the fields of an  $n$ -tuple and access the fields by name. Records are pretty much like structs in C.

$$\{l_1 : \tau_1, \dots, l_n : \tau_n\} \quad (n \geq 0)$$

This is a record type with labels  $l_1$  to  $l_n$ , where each label  $l_i$  corresponds to a particular type  $\tau_i$ . For brevity we will represent record types using the following syntax:

$$\{l_i : \tau_i\}_{i=1}^n$$

and also use similar syntax for similar types/expressions.

## 5.1 Static Semantics

$$\frac{[\Gamma \vdash e_i : \tau_i]_{i=1}^n}{\Gamma \vdash \{l_i = e_i\}_{i=1}^n : \{l_i : \tau_i\}_{i=1}^n} \text{RECORD-I} \qquad \frac{\Gamma \vdash e : \{l_i : \tau_i\}_{i=1}^n}{\Gamma \vdash e.l_k : \tau_k} \text{RECORD-E}$$

Note that the order of fields in the record type do not matter; two types with identical labels and types for labels but different orderings of types are considered equivalent. The same applies for record expressions.

**Example 5.1.**

$$\begin{aligned} \{a : \tau_1, b : \tau_2\} &\equiv \{b : \tau_2, a : \tau_1\} \\ \{a = 1, b = \langle 2, 3 \rangle\} &\equiv \{b = \langle 2, 3 \rangle, a = 1\} \end{aligned}$$

## 5.2 Dynamic Semantics

$$\frac{[e_i \text{ value}]_{i=1}^n}{\{l_i = e_i\}_{i=1}^n.l_i \mapsto e_i} \qquad \frac{e_1 \text{ value} \quad \dots \quad e_{i-1} \text{ value} \quad e_i \mapsto e'_i}{\{l_1 = e_1, \dots, l_i = e_i, \dots, l_n = e_n\} \mapsto \{l_1 = e_1, \dots, l_i = e'_i, \dots, l_n = e_n\}}$$

The second rule in the dynamic semantics stipulates a left-to-right evaluation of the components of the record. Also note that the record is eagerly evaluated; it could be implemented lazily as well.

## 5.3 Objects

Consider mutually recursive functions in ML:

```
fun even (x : nat) : bool =
  case x of
    zero      => inl()
  | succ(x') => not (odd x')

and odd (x : nat) : bool =
  case x of
    zero      => inr()
  | succ(x') => not (even x')
```

Note that `inl()` is the boolean value `true` and `inr()` is the boolean value `false`, as described previously. Mutual recursion can be expressed using the `fix` operator and records like this:

$$\begin{aligned} EO &:= \text{fix}(eo : \{\text{even} : \text{nat} \rightarrow \text{bool}, \text{odd} : \text{nat} \rightarrow \text{bool}\}). \\ &\quad \{\text{even} = \lambda x : \text{nat}. \text{case} \dots eo. \text{odd} \dots, \text{odd} = \lambda x : \text{nat}. \text{case} \dots eo. \text{even} \dots\} \\ &\mapsto \{\text{even} = \lambda x : \text{nat} \dots (\text{fix} \dots). \text{odd}, \text{odd} = \lambda x : \text{nat} \dots (\text{fix} \dots). \text{even}\} \end{aligned}$$

Suppose we rename `eo` to `this`. Then, `this.even` is:

$\lambda x : \text{nat} . \underline{\text{case}} \dots \text{this} . \text{odd} \dots$

and the same for *this.odd*. This looks identical to how we would access object methods in object-oriented languages such as Java, and indeed, *EO* is precisely such an *object*. Thus, we can think of objects as self-referential records through the use of fix.

## 6 Variants

*Variants* are to sums as records are to products: they are labelled sums. These are the types that are created in ML using the `datatype` declaration.

### 6.1 Static Semantics

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash [l_i = e_i]_{[l_i : \tau_i]_{i=1}^n} : [l_i : \tau_i]_{i=1}^n} \text{ VARIANT-I} \qquad \frac{\Gamma \vdash e : [l_i : \tau_i]_{i=1}^n \quad [\Gamma, x_i : \tau_i \vdash e_i : \tau]_{i=1}^n}{\Gamma \vdash \underline{\text{case}} e \{ [l_i = x_i : \tau_i] = e_i \}_{i=1}^n : \tau} \text{ VARIANT-E}$$

Note that in the intro form we annotate the expression for typechecking purposes as in the sum type. The elim form consists of an n-way case analysis over the variant  $e$ . Just as with the binary sum, all branches of the case analysis must yield a result with the same type  $\tau$ .

We can actually express the binary sum type using variants:

$$\begin{aligned} \tau_1 + \tau_2 &\cong [\underline{\text{inl}} : \tau_1, \underline{\text{inr}} : \tau_2] \\ \underline{\text{inl}}(e) &\cong [\underline{\text{inl}} = e] \\ \underline{\text{inr}}(e) &\cong [\underline{\text{inr}} = e] \end{aligned}$$

The same is true for booleans:

$$\text{bool} \cong [\text{true}, \text{false}]$$

## 7 Recursive Types

Consider the type `nat`:

$$\frac{}{\text{zero} : \text{nat}} \qquad \frac{n : \text{nat}}{\text{succ}(n) : \text{nat}}$$

From the rules, we can imagine that `nat` is a sum type of a form something like the following:

$$\text{nat} \cong 1 + \text{nat}$$

since `nat` is either zero or the successor of another `nat`. Here  $\cong$  is isomorphism. We wish to solve this type equation up to isomorphism.

Given the above equation, we can think of two different functions that can be performed on the types `nat` and `1 + nat`:

$$\begin{aligned} \underline{\text{roll}} &: (1 + \text{nat}) \rightarrow \text{nat} \\ \underline{\text{unroll}} &: \text{nat} \rightarrow (1 + \text{nat}) \end{aligned}$$

It seems that there is only one way to write roll<sup>5</sup>:

$$\underline{\text{roll}} = \lambda x : 1 + \text{nat} . \underline{\text{case}}(x, \underline{\text{inl}}(\langle \rangle) \Rightarrow z \langle \rangle \mid \underline{\text{inr}}(x') \Rightarrow s x')$$

where  $z : 1 \rightarrow \text{nat}$  and  $s : \text{nat} \rightarrow \text{nat}$  are some functions.

Since this is the only way to express roll, we can think of the functions  $z$  and  $s$  as “defining” roll. So:

$$(1 + \text{nat}) \rightarrow \text{nat} \cong (1 \rightarrow \text{nat}) \times (\text{nat} \rightarrow \text{nat})$$

and we can construct a value of the type of the right-hand side:

$$\{\text{zero} : 1 \rightarrow \text{nat}, \text{succ} : \text{nat} \rightarrow \text{nat}\}$$

Thus we can express **nat** using the type equation given above, and the functions **zero** and **nat** with the types specified.

Similarly, lists (of natural numbers) can be expressed using this type equation:

$$\text{list} \cong 1 + (\text{nat} \times \text{list})$$

and trees:

$$\text{tree} \cong 1 + (\text{tree} \times \text{nat} \times \text{tree})$$

## 7.1 Solving the Equations

We wish to be able to solve the type equations give above. Notice that all of them have a particular form:

$$X \cong F(X)$$

where  $X$  is the type, and  $F$  is a function on the type  $X$ . Then, it appears that what we really need is a fixed point for  $F$ , which will yield us the solution to the equation.

Thus we define a fixed point generator for  $F$ ,  $\mu$ :

$$\mu(F) \cong F(\mu(F))$$

This fixed point generator then allows us to properly define recursive types. We use the syntax:

$$\mu X. \tau$$

where  $X$  is a type variable, bound within  $\tau$ . So for example:

$$\begin{aligned} \text{nat} &:= \mu N. 1 + N \\ \text{list} &:= \mu L. 1 + (\text{nat} \times L) \end{aligned}$$

## 7.2 Static Semantics

$$\frac{\Gamma \vdash e : [\mu X. \tau / X] \tau}{\Gamma \vdash \underline{\text{roll}}_{\mu X. \tau}(e) : \mu X. \tau} \text{ROLL} \qquad \frac{\Gamma \vdash e : \mu X. \tau}{\Gamma \vdash \underline{\text{unroll}}(e) : [\mu X. \tau / X] \tau} \text{UNROLL}$$

roll has type annotations for typechecking purposes.

---

<sup>5</sup>Not entirely true, though.

### 7.3 Dynamic Semantics

$$\frac{\text{roll}(e) \text{ value}}{\text{unroll}(\text{roll}(e)) \mapsto e}$$

$$\frac{e \mapsto e'}{\text{unroll}(e) \mapsto \text{unroll}(e')}$$

We can “get back” `nat` using recursive types:

$$\begin{aligned} \text{nat} &:= \mu N. 1 + N \\ \text{zero} &:= \text{roll}(\text{inl}(\langle \rangle)) \\ \text{succ}(x) &:= \text{roll}(\text{inr}(x)) \quad \text{where } x : \text{nat} \\ \text{natcase}(x : \text{nat}, e_0, e_1) &:= \text{case}(\text{unroll}(x : \text{nat}), \text{inl}(-) \Rightarrow e_0 \mid \text{inr}(x' : \text{nat}) \Rightarrow e_1) \end{aligned}$$

**Example 7.1.** We can also express lists in the same way:

$$\begin{aligned} \text{list} &:= \mu L. 1 + (\text{nat} \times L) \\ \text{nil} &:= \text{roll}(\text{inl}(\langle \rangle)) \\ \text{cons}(n, l) &:= \text{roll}(\text{inr}(n, l)) \end{aligned}$$

Interestingly, we can see a parallel between lists expressed as such and linked list using pointers in a language like C. The use of `roll` and `unroll` is analogous to pointer referencing and dereferencing.