Lecture Notes on Mutable Memory

15-814: Types and Programming Languages
Frank Pfenning

Lecture 22
Tuesday, November 19, 2019

1 Introduction

We have moved from a semantics directly on expressions to one that makes memory explicit and supports concurrency (at the discretion of the scheduler or the language designer). Memory is allocated and then written to at most once; after that it may be read many times.

In imperative languages we can also mutate the contents of a memory cell by writing a different value to it. In a functional language, this is typically segregated, either in a monad (as in Haskell) or via a new type of mutable references (as in ML). We pursue here the latter approach because there is a slightly lower conceptual overhead.

2 The Type of Mutable References

We introduce one new type constructor and three new forms of expression into our functional language:

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \ldots \mid \mathsf{ref}\ \tau \\ \text{Expressions} & e & ::= & \ldots \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid\ !e \end{array}$$

Operationally, $\mathsf{ref}\ e$ evaluates e to a value v, then create a new mutable reference m and initializes its value to v. $e_1 := e_2$ evaluates e_1 to a mutable reference m, then e_2 to a value v_2 and stores v_2 in m. It returns just the unit element, since its principal task is the effect on m. Finally, $!e$ (which has nothing to do with ! to denote persistent semantic objects) evaluates e to a

reference m and returns the current value of m. Based on this description, we type these new expressions as follows

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau}\ \mathsf{ref/create} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}\ \mathsf{ref/write}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash\ !e : \tau}\ \mathsf{ref/read}$$

These rules do not fit the previous patterns of constructor and destructors because of the rule for mutation ref/write.

It seems difficult, if not impossible, to specify the semantics of mutable references directly on expressions in the style we have done before. Fortunately, we already have a semantics with an explicit store so we can update that. The textbook instead generalizes the small-step semantics for expression by adding a single store μ and now stepping $\mu \parallel e \mapsto \mu' \parallel e'$ [Har16, Chapters 34 & 35].

3 Translation to Our Concurrent Language

We exploit the fact we already have a representation of memory in this translation, and only two small twists are necessary. Warning: some of what is below we will later find out is not quite right. We write m for the address of a mutable cell.

$$\begin{array}{l}[\![\mathsf{ref}\ e]\!]\, d = m \leftarrow [\![e]\!]\, m\ ; \\ \qquad\qquad\quad d^W.\mathsf{addr}(m)\end{array}$$

Here, we introduce a new form of value, $\mathsf{addr}(m)$ which denotes the address of a mutable cell, here m. This value is deposited in destination d as required.

Reading from a mutable destination is simple.

$$\begin{array}{l}[\![!e]\!]\, d = d_1 \leftarrow [\![e]\!]\, d_1\ ; \\ \qquad\quad \mathsf{case}\ d_1^R\ (\mathsf{addr}(m) \Rightarrow d^W \leftarrow m^R)\end{array}$$

Finally, mutating a cell. At first we might try

$$\begin{array}{lll}[\![e_1 := e_2]\!]\, d = d_1 \leftarrow [\![e_1]\!]\, d_1\ ; & & \\ \qquad\qquad\quad d_2 \leftarrow [\![e_2]\!]\, d_2\ ; & & \\ \qquad\qquad\quad \mathsf{case}\ d_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \leftarrow d_2^R) & \% \textit{ bug here!} & \end{array}$$

The problem here is that the translation of $e_1 := e_2$ is supposed to write to destination d, but does not do so. Recall that we decreed that the assignment should return the unit element, so we might write

$$\begin{aligned}[\![e_1 := e_2]\!]\, d = {} & d_1 \leftarrow [\![e_1]\!]\, d_1 \;; \\ & d_2 \leftarrow [\![e_2]\!]\, d_2 \;; \\ & \mathsf{case}\ d_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \leftarrow d_2^R \;; \\ & \qquad\qquad\qquad\qquad d.\langle\,\rangle)\end{aligned}$$

However, this requires a version of the copy process that allows a continuation. Let's write this as $m^W \Leftarrow d_2^R$, and we get

$$\begin{aligned}[\![e_1 := e_2]\!]\, d = {} & d_1 \leftarrow [\![e_1]\!]\, d_1 \;; \\ & d_2 \leftarrow [\![e_2]\!]\, d_2 \;; \\ & \mathsf{case}\ d_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \Leftarrow d_2^R \;; \\ & \qquad\qquad\qquad\qquad d.\langle\,\rangle)\end{aligned}$$

The new process expression has the dynamics

$$!\mathsf{cell}\ m\ W, !\mathsf{cell}\ c\ W', \mathsf{proc}\ d\ (m^W \Leftarrow c^R\ ;\ P) \mapsto !\mathsf{cell}\ m\ W', \mathsf{proc}\ d\ P \quad \%\ \textit{bug!}$$

Writing this out, however, we notice a second problem: the cell m has to be ephemeral. If it were persistent, then after this transition m would have two values: W and W'.

We can fix this in two ways. Either we make all cells (mutable or not) ephemeral. This means we have to revisit all the rules so far and make sure cell are not consumed when they are read but carried over. Alternatively, we can make only mutable cells ephemeral and keep all others persistent. Let's use the first approach. We modify the rules at the end of Section L21.4 by dropping the ! everywhere. Where we match against $!\mathsf{cell}\ c\ W$ on the left-hand side, we just replace it by $\mathsf{cell}\ c\ W$ and repeat it on the right-hand side. The rule for the new "write" construct becomes

$$\mathsf{cell}\ m\ W, \mathsf{cell}\ c\ W', \mathsf{proc}\ d\ (m^W \Leftarrow c^R\ ;\ P) \mapsto \mathsf{cell}\ m\ W', \mathsf{cell}\ c\ W', \mathsf{proc}\ d\ P$$

For the other approach, see Exercise 1.

4 Race Conditions

In the presence of mutable references, sequential computation proceeds as before, scheduling such that in $x \leftarrow P\ ;\ Q$ the process P completes (and therefore writes to x) before Q starts. This also means that the read and write operations on mutable cells have a well-defined order.

Under the concurrent semantics, however, the picture is more complicated. Consider the following expression:

$(\lambda x.\, \langle x := \mathit{succ}\ x, \langle x := \mathit{succ}\ x, !x\rangle\rangle)\ (\mathsf{ref}\ \mathit{zero})$

The value of this expression will be

$\langle\langle\rangle, \langle\langle\rangle, n\rangle\rangle$

where $n \in \{\overline{0}, \overline{1}, \overline{2}\}$. For example, we obtain $\overline{0}$ if $!x$ executes before the increments. Note also that either increment or dereference of the value might have to wait until the initialization of the mutable cell with $\overline{0}$ completes because the body of the function can execute in parallel with the argument.

Despite these difficulties, progress and preservation theorems continue to hold, but it becomes much more difficult to reason about the correctness of programs. Similarly, we don't lose all of parametricity, but logical equality (and, more generally, logical relations) now require *step-indexing* [AM01, TTA+13].

5 Linearity

Now that we (provisionally) decided to make all cells ephemeral, we can wonder if we really need to carry over all cells during a transition, or if we may be able to drop some if they can no longer be accessed. As an example, let's examine the translation for function application.

$$\begin{aligned}[\![e_1\ e_2]\!]\ d = {} & d_1 \leftarrow [\![e_1]\!]\ d_1\ ; \\ & d_2 \leftarrow [\![e_2]\!]\ d_2\ ; \\ & d_1^R.\langle d_2, d\rangle\end{aligned}$$

The destination d_1 will be written by the translation $[\![e_1]\!]\ d_1$ and is then read by the last line. But it could not be used beyond that because it can not occur elsewhere in the program since d_1 is fresh not passed to anywhere.

The situation is different for d_2. Even though it is freshly allocated here it is passed on to the function stored in d_1 so it "escapes its lexical scope" and we cannot deallocate it here.

Methodologically, we should now examine various constructs to see which destinations we may be able to "deallocate" by not copying them from the left-hand sides of transition rule to the right. But this is complicated, so first we examine what would be required so that we would *never* have to copy cells that are being read from (excluding mutable cells from consideration for the moment, for simplicity). Essentially, can we delineate

a subsect of the language so that every cell will not only be *written to* once, but also *read from* once. Of course, as you might expect in this course after all we have been through together, this is expressed as a type system! Every memory cell will have not only a unique provider (to write it) but also a unique client (to read from it). We call a type system that enforces this property *linear*, after Girard's *linear logic* [Gir87].

We start with some intuition. We say a *function* is linear in one of its arguments if it uses that argument exactly once. The notion of "usage" here is a dynamic one; it doesn't mean that the variable *occurs* exactly once, as we will see.

$$\lambda x.\, x \qquad \text{(linear)}$$

This is linear in x and therefore the whole expression is linear.

$$\lambda x.\, \lambda y.\, x \qquad \text{(not linear)}$$

This expression is linear in x but not linear in y and therefore not linear. It's not linear in y because y is not used, but linearity requires a single use. Related to linearity is the is the notion of *affine*. A function is *affine* in a variable if it is used *at most once*. So the function above is *affine* but not *linear*. The notion of affine has recently received a lot of attention because the Rust programming language treats memory references as affine.

$$\lambda x.\, \langle x, x \rangle \qquad \text{(not linear)}$$

This expression is not linear because x is used twice and hence more than once. Functions that use their argument *at least once* are called *strict*. The notion of strictness is important because it is useful in the optimization of call-by-need languages such as Haskell. If we have a function application $e_1\, e_2$ and we can tell that e_1 denotes a *strict* function we can safely evaluate e_2 rather than waiting until e_1 might need its argument.

$$\lambda x.\, \mathsf{if}\ x\ \mathit{false}\ x \qquad \text{(not linear)}$$

This function is not linear in x. It uses x the first time to decide the condition, and then again when x is false. However, if x is false this returns x which is *false*, so extensionally equal would be

$$\lambda x.\, \mathsf{if}\ x\ \mathit{false}\ \mathit{false} \qquad \text{(linear)}$$

which is linear. These two examples show that linearity is an intensional property of expressions (how do they compute) and not an extensional property (what do they compute).

$$\lambda x.\, \lambda y.\, \mathsf{if}\ x\ y\ (\mathit{not}\ y) \qquad \text{(linear)}$$

This function is linear: x is used once as subject of the conditional. The variable y occurs twice, but whenever this expression is executed it is used exactly once: if y is true then in the first branch, and if y is false then in the second branch.

$$\lambda x.\, (\lambda y.\langle\,\rangle)\, x \qquad \text{(not linear)}$$

It shouldn't be suprising by now that this is not linear, since y is not linear in $\langle\,\rangle$. But, moreover, the whole expression is not linear in x, even though x occurs exactly once. That's because x occurs in a position where it will be dropped. On the other hand

$$\lambda x.\, (\lambda y.\, \langle y, \langle\,\rangle\rangle)\, x \qquad \text{(linear)}$$

6 Linear Typing of Expressions

With these examples, we now work through the inference rules for expressions and classify those that are linear. We use a different notation for functions, eager pairs, sums, etc. since the connectives are subtly different from the regular ones. Our judgment has the form

$$\Delta \vdash e : \tau$$

where Δ is a context of variables, each of which must be used once in e.

Linear Functions $\tau \multimap \sigma$. A function is linear just if its parameter is used linearly in its body.

$$\frac{\Delta, x : \tau \vdash e : \sigma}{\Delta \vdash \lambda x.\, e : \tau \multimap \sigma}\ \multimap I$$

When applying a function we have to divide up the variables among those that occur in the function (Δ_1) and those that occur in the argument (Δ_2).

$$\frac{\Delta_1 \vdash e_1 : \tau_2 \multimap \tau_1 \quad \Delta_2 \vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \vdash e_1\, e_2 : \tau_1}\ \multimap E$$

Our usual presupposition regarding contexts kicks in and we implicit require the $\mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) = \emptyset$.

When we look up variables, there cannot be other variables in the context because they would not be used and therefore not be linear.

$$\frac{}{x : \tau \vdash x : \tau}\ \mathsf{var}$$

Eager Linear Pairs $\tau \otimes \sigma$. Eager linear pairs are written as $\tau \otimes \sigma$. The rules are straightforwardly patterned after previous rules, keeping in mind that for the destructor (case), the variables standing for the components of the pair must be linear.

$$\frac{\Delta_1 \vdash e_1 : \tau_1 \quad \Delta_2 \vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2} \otimes I \qquad \frac{\Delta \vdash e : \tau_1 \otimes \tau_2 \quad \Delta', x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Delta, \Delta' \vdash \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \otimes E$$

The nullary version of pairs, the unit is written as 1 and the rules are the nullary version of the binary rules above (see Section 7).

Linear Sums $\tau \oplus \sigma$. Actually, we will show the labeled, n-ary version $\oplus_{i \in I}(i : \tau_i)$. In the constructor rule $\oplus I$, there is not much to consider.

$$\frac{\Delta \vdash e : \tau_j}{\Delta \vdash j \cdot e : \oplus_{i \in I}(i : \tau_i)} \oplus I$$

For the destructor (case) we need to consider the same as for the conditional if in the last section: only one branch of the case will be taken, so all branches must be checked with the same linear context.

$$\frac{\Delta \vdash e : \oplus_{i \in I}(i : \tau_i) \quad (\text{for all } i \in I) \quad \Delta', x : \tau_i \vdash e'_i : \tau'}{\Delta, \Delta' \vdash \mathsf{case}\ e\ (i \cdot x \Rightarrow e'_i)_{i \in I} : \tau'} \oplus E$$

Recursion. The remaining type constructors follow similar patterns so we omit the details (see Section 7 for a listing). Recursion, however, is interesting. The computation rule for fixed points it

$$\mathsf{fix}\ f.\, e \mapsto [\mathsf{fix}\ f.\, e/f]e$$

This already departed from the pattern of the other rules. For one, we substitute an expression ($\mathsf{fix}\ f.\, e$) for a variable f in an expression e, while all the other rules just substitute values for variables. For another, it is not attached to a particular type constructor and can always be applied.

There are several sources of operational "nonlinearity" in this rule. First, even if f occurs only once in e, it is replaced by another expression ($\mathsf{fix}\ f.\, e$) containing e, thereby duplicating e. Also, when we define a recursive function we would like to make multiple recursive calls and still consider the function linear.

For example, the function that takes a bit string (usually considered just a binary number) and flips every bit (see Lecture L20.5) should be linear:

each bit of the input string is read and a corresponding bit written to the output.

$$\mathit{bits} \simeq (\mathtt{b0} : \mathit{bits}) \oplus (\mathtt{b1} : \mathit{bits}) \oplus (\mathtt{e} : 1)$$

$$\mathit{flip} : \mathit{bits} \multimap \mathit{bit}$$

$$\begin{array}{ll}\mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\ \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{e} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot y)\ )\end{array}$$

Note that there is no recursive call to *flip* in the third branch and yet we should consider the function linear. In order to formally represent this, we have to add a second, nonlinear context to our judgment and populate it with recursively defined variables. Moreover, since the body of the recursively defined expression is duplicated when it is unwound, it may not depend on any linear variables.

$$\frac{\Gamma, f : \tau\ ;\ \cdot \vdash e : \tau}{\Gamma\ ;\ \cdot \vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{rec}$$

With these rules (and the straightforward ones for *fold* and *unfold*) the *flip* function can indeed be checked as linear.

Now we need to generalize all the other rules, adding the nonlinear context Γ and propagating it to all premises, allowing it for variables, etc. You can find a listing in Section 7.

This example is also remarkable because a tiny change in the last branch of the conditional

$$\mathit{flip} : \mathit{bits} \multimap \mathit{bit}$$

$$\begin{array}{ll}\mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\ \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{e} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot \langle\rangle)\ ) & \%\ \text{bug here!}\end{array}$$

makes this function now nonlinear: y is not used. Besides the code shown earlier, we can also fix the problem by *using* $y : 1$.

$$\mathit{flip} : \mathit{bits} \multimap \mathit{bit}$$

$$\begin{array}{ll}\mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\ \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{e} \cdot y \Rightarrow \mathsf{case}\ y\ (\langle\rangle \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot \langle\rangle))\ )\end{array}$$

7 Linear Rule Summary

The syntax for the language of expression does not change, but the language of types is new.

$$\text{Linear types} \quad \tau \quad ::= \quad \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid 1 \mid \oplus_{i \in I}(i : \tau_i) \mid \rho\alpha.\, \tau \mid \alpha$$

The definition of values and the rules for evaluation remain the same as for our nonlinear functional language.

We name the propositional rules I (for introduction, representing a constructor for a type) and E (for elimination, representing a destructor for a type). Missing here are lazy pairs (left for a future lecture) and the universal and existential quantifiers. The latter are orthogonal, but would introduce even further syntactic overhead since we would have to track type variables in addition to ordinary linear variables and recursion variables.

$$\frac{}{\Gamma\,;\, x : \tau \vdash x : \tau}\ \mathsf{var} \qquad \frac{f : \tau \in \Gamma}{\Gamma\,;\, \cdot \vdash f : \tau}\ \mathsf{recvar} \qquad \frac{\Gamma, f : \tau\,;\, \cdot \vdash e : \tau}{\Gamma\,;\, \cdot \vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{rec}$$

$$\frac{\Gamma\,;\, \Delta, x : \tau \vdash e : \sigma}{\Gamma\,;\, \Delta \vdash \lambda x.\, e : \tau \multimap \sigma} \multimap I \qquad \frac{\Gamma\,;\, \Delta_1 \vdash e_1 : \tau_2 \multimap \tau_1 \quad \Gamma\,;\, \Delta_2 \vdash e_2 : \tau_2}{\Gamma\,;\, \Delta_1, \Delta_2 \vdash e_1\, e_2 : \tau_1} \multimap E$$

$$\frac{\Gamma\,;\, \Delta_1 \vdash e_1 : \tau_1 \quad \Gamma\,;\, \Delta_2 \vdash e_2 : \tau_2}{\Gamma\,;\, \Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2} \otimes I \qquad \frac{\Gamma\,;\, \Delta \vdash e : \tau_1 \otimes \tau_2 \quad \Gamma\,;\, \Delta', x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma\,;\, \Delta, \Delta' \vdash \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \otimes E$$

$$\frac{}{\Gamma\,;\, \cdot \vdash \langle\rangle : 1}\ 1I \qquad \frac{\Gamma\,;\, \Delta \vdash e : 1 \quad \Gamma\,;\, \Delta' \vdash e' : \tau'}{\Gamma\,;\, \Delta, \Delta' \vdash \mathsf{case}\ e\ (\langle\rangle \Rightarrow e') : \tau'}\ 1E$$

$$\frac{(j \in I) \quad \Gamma\,;\, \Delta \vdash e : \tau_j}{\Gamma\,;\, \Delta \vdash j \cdot e : \oplus_{i \in I}(i : \tau_i)} \oplus I$$

$$\frac{\Gamma\,;\, \Delta \vdash e : \oplus_{i \in I}(i : \tau_i) \quad (\text{for all } i \in I) \quad \Gamma\,;\, \Delta', x : \tau_i \vdash e_i' : \tau'}{\Gamma\,;\, \Delta, \Delta' \vdash \mathsf{case}\ e\ (i \cdot x \Rightarrow e_i')_{i \in I} : \tau'} \oplus E$$

$$\frac{\Gamma\,;\, \Delta \vdash e : [\rho\alpha.\, \tau/\alpha]\tau}{\Gamma\,;\, \Delta \vdash \mathsf{fold}\ e : \rho\alpha.\, \tau} \rho I \qquad \frac{\Gamma\,;\, \Delta \vdash e : \rho\alpha.\, \tau}{\Gamma\,;\, \Delta \vdash \mathsf{unfold}\ e : [\rho\alpha.\, \tau/\alpha]\tau} \rho E$$

Exercises

Exercise 1 Provide an alternative dynamics for our language with mutable cells, where regular cells become persistent once written, while mutable

cells are ephemeral. You may have to introduce some new kinds of semantic objects or some new forms of process expression, or both.

Exercise 2 Write a linear increment function on natural numbers in binary representation.

Exercise 3 Recall the definition of a purely positive type, updated to reflect the notation for linear types.

$$\tau^+ ::= 1 \mid \tau_1^+ \otimes \tau_2^+ \mid \oplus_{i \in I}(i : \tau_i^+) \mid \rho\alpha^+.\, \tau^+ \mid \alpha^+$$

Even in the purely linear language, it is possible to *copy* a value of purely linear type. Define a family of functions

$$\mathit{copy}_{\tau^+} : \tau^+ \multimap (\tau^+ \otimes \tau^+)$$

such that $\mathit{copy}_{\tau^+}\ v \mapsto^* \langle v, v \rangle$ for every $v : \tau^+$. You do not need to prove this property, just give the definitions of the *copy* functions. Your definitions may be mutually recursive.

Exercise 4 A type isomorphism is *linear* if the functions *Forth* and *Back* are both linear. For each of the following pairs of types provide linear functions witnessing an isomorphism if they exist, or indicate no linear isomorphism exists. You may assume all functions terminate and use either extensional or logical equality as the basis for your judgment.

1. $\tau \multimap (\sigma \multimap \rho)$ and $\sigma \multimap (\tau \multimap \rho)$
2. $\tau \multimap (\sigma \multimap \rho)$ and $(\tau \otimes \sigma) \multimap \rho$
3. $\tau \multimap (\sigma \otimes \rho)$ and $(\tau \multimap \sigma) \otimes (\tau \multimap \rho)$
4. $(\tau \oplus \sigma) \multimap \rho$ and $(\tau \multimap \rho) \otimes (\sigma \multimap \rho)$
5. $(1 \oplus 1) \multimap \tau$ and $\tau \otimes \tau$

Exercise 5 Write out the following theorems, updated to the purely linear language (where only recursively defined variables are nonlinear). We change neither the definition of value nor the rules for stepping from our previous language that does not employ linearity.

1. Canonical forms for types $\multimap$, $\otimes$, 1, $\oplus$, and ρ. No proofs are needed.
2. The substitution properties, in a form sufficient needed for preservation. No proofs are needed.

3. The preservation property for evaluation of closed linear expressions. Show the proof cases for linear functions.

4. The progress property for closed linear expressions. Show the proof cases for linear functions.

5. Where do these properties and their proofs differ when compared to our language that does not enforce linearity?

References

[AM01] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

[TTA+13] Aaron Joseph Turon, Jacob Thamsbord, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *Symposium on Principles of Programming Languages (POPL'13)*, pages 343–356, Rome, Italy, January 2013. ACM.