Linear Type Systems for Communication

Tom Murphy VII

18 Dec 2001

Abstract

Enhancing the type system of a programming language brings benefits on many fronts. We define and explore a simple concurrent functional language in a linear typing discipline, showing how more powerful types can lead to more controlled interaction between threads.

1 Introduction

A type system serves many purposes in a programming language. A type system excludes dangerous programs. It may make implementations more efficient. Type systems can be *descriptive*, helping to explain how or what code does. Types may also be *prescriptive*, enforcing the manner in which code is used by a client. A more sophisticated type system typically leads to benefits in all of these categories, by allowing types to be more expressive.

This project is primarily interested in the *prescriptive* nature of types. By enhancing the type system of a simple concurrent functional language with linear types, I show how it is possible to enforce precise communication protocols in client code.

This paper consists of sections as follows: First, an introduction to linear types and the LinML programming language with an emphasis on programmer's intuition; a formal description of LinML's type system and dynamic semantics; a series of examples of how linear types can be leveraged for a concurrent functional language; and a discussion of related work. An SML implementation of a LinML typechecker and interpreter are included as an appendix.

2 LinML and Linear Types

LinML is a an extension of MinML [1] (a modest subset of the SML core language) with concurrency primitives and linear types.

The intuition behind linearity is that variables of linear type must be used exactly once.

One way for such variables to be introduced is as an argument to a linear function (whose type is written with the $\multimap$ symbol):

```
f1 ≜ lfn x : bool => x
f2 ≜ lfn x : bool => x andalso x
f3 ≜ lfn x : bool => true
f4 ≜ lfn x : bool => if true then x else x
```

f_1 has linear type `bool -o bool`, as it uses its argument x exactly once. f_2 and f_3 are ill-typed, as they use x too many and too few times, respectively. Though x appears twice in f_4, intuition tells us that exactly one will be actually used (this is the case even if the condition expression is not trivial); f_4 also has linear type `bool -o bool`.

The linear typing discipline leads to new interpretations of other familiar type constructs. For instance, all the fields of a record (tuple) must be used exactly once:

```
t1 ≜ {l1 = true,
      l2 = lfn x : bool => x,
      l3 = false}

t2 ≜ lett {l1 = a, l2 = b, l3 = c} = t1
     in
          if a then b ^ c else c andalso b ^ true
     end
```

Here, `^` stands for application of a linear function. t_1 is a tuple expression and has type `{l1 : bool, l2 : bool -o bool, l3 : bool}`. `lett` breaks apart a tuple, and each variable (a, b, c) must be used exactly once in the body.

In a lazy record, the programmer must choose exactly one field to project:

```
l1 ≜ lfn x : bool => <l1 = x, l2 = not x>
l2 ≜ #l2 (l1 ^ true)
```

l_1 has type `<l1 : bool, l2 : bool>`. `#`*fieldname* chooses a field from a lazy record. Since the programmer will ultimately only choose one field, it is acceptable to have linear variables re-used in each field. In fact, all fields must use exactly the same set of linear variables.[1] None of the fields are evaluated until one is projected out.

Sums work essentially as they do in MinML:

[1] LinML leaves out the $\top$ connective, since it did not seem very useful for these kinds of programs and it complicates type-checking, especially with n-ary labelled records and sums.

```
s1 ≜ inj(SOME, [SOME : bool, NONE : {}], true)
s2 ≜ case_v s1 of
            SOME => if v then print "Yes" else print "No"
          | NONE => v
```

s_1 injects `true` into the `SOME` branch of the sum, and has type `[SOME : bool, NONE : {}]`. The `case` construct binds the same variable v in all arms. (Like lazy tuples, all arms must use the same set of resources since exactly one will be chosen.) `print` outputs a string and returns the empty record.

It is possible to program in the standard unrestricted style by using the `exp` operator. If an expression uses no linear resources, then it can be bound to a variable that can be used as many times as desired:

```
b1 ≜ exp (lfn x : bool => x)
b2 ≜ use id = b1
     in id (id true)
     end
```

b_1 has type `!(bool -o bool)`. *id* can be used arbitrarily often in the body of the `use` expression.

Finally, LinML has primitives for concurrent programming:

```
c1 ≜ chan string
c2 ≜ lett {r=rc, s=sc} = c1
     in spawn (lfn x : {} =>
                 lett {data = s, channel = c} = recv rc
                 in print s;
                    close c;
                    x
                 end);
        let ns = send (sc, "Hello")
        in close ns
        end
     end
```

c_1 has type `{s = schan string, r = rchan string}`; it returns a pair of channels, one for sending and one for receiving. `spawn` takes a function of type `{} -o {}` and creates a new process running in parallel with the current process. The process that is spawned here takes the receiving end of the channel (rc) with it. `recv` takes an `rchan` τ and returns a tuple `{data :` τ`, channel : rchan` τ`}`. Since channels can only be used once, `recv` also returns a new channel for the next datum. `close` shuts down a receiving or sending channel. Finally, `send` takes an `schan` τ and a τ and returns a new channel. In the example above, the original thread sends the message `"Hello"` to the newly spawned thread, which prints it.

3 Type System

This section gives a formal description of the type system.

$$
\begin{array}{rl}
\tau ::= & \texttt{bool} \mid \texttt{string} \mid \texttt{rchan}\ \tau \mid \texttt{schan}\ \tau \\
\mid & \tau_1\ \texttt{-o}\ \tau_2 \mid \tau_1\ \texttt{->}\ \tau_2 \\
\mid & \texttt{!}\tau \mid \forall\ t.\tau \mid t \\
\mid & \{l_1\ \texttt{=}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ l_n\ \texttt{=}\ \tau_n\} \quad (\texttt{n} \neq \texttt{1}) \\
\mid & \texttt{<}l_1\ \texttt{=}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ l_n\ \texttt{=}\ \tau_n\texttt{>} \quad (\texttt{n} > \texttt{1}) \\
\mid & \texttt{[}l_1\ \texttt{=}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ l_n\ \texttt{=}\ \tau_n\texttt{]} \quad (\texttt{n} \geq \texttt{1})
\end{array}
$$

A type is well-formed (ok) if it does not contain free variables and does not have duplicate labels:

$$\frac{}{\Theta \vdash \texttt{bool}\ \mathsf{ok}} \qquad \frac{}{\Theta \vdash \mathit{string}\ \mathsf{ok}}$$

$$\frac{\Theta \vdash \tau\ \mathsf{ok}}{\Theta \vdash \texttt{rchan}\,\tau\ \mathsf{ok}} \qquad \frac{\Theta \vdash \tau\ \mathsf{ok}}{\Theta \vdash \texttt{schan}\,\tau\ \mathsf{ok}}$$

$$\frac{\Theta \vdash \tau_1\ \mathsf{ok} \quad \Theta \vdash \tau_2\ \mathsf{ok}}{\Theta \vdash \tau_1 \multimap \tau_2\ \mathsf{ok}} \qquad \frac{\Theta \vdash \tau_1\ \mathsf{ok} \quad \Theta \vdash \tau_2\ \mathsf{ok}}{\Theta \vdash \tau_1 \to \tau_2\ \mathsf{ok}}$$

$$\frac{\Theta \vdash \tau\ \mathsf{ok}}{\Theta \vdash !\tau\ \mathsf{ok}} \qquad \frac{t \in \mathtt{dom}(\Theta)}{\Theta \vdash t\ \mathsf{ok}}$$

$$\frac{\Theta, t \vdash \tau\ \mathsf{ok}}{\Theta \vdash \forall t.\tau\ \mathsf{ok}}$$

$$\frac{\Theta \vdash \tau_1\ \mathsf{ok} \quad \ldots \quad \Theta \vdash \tau_n\ \mathsf{ok} \quad i \neq j \supset l_i \neq l_j}{\Theta \vdash \{l_1 = \tau_1, \ldots, l_n = \tau_n\}\ \mathsf{ok}}$$

$$\frac{\Theta \vdash \tau_1\ \mathsf{ok} \quad \ldots \quad \Theta \vdash \tau_n\ \mathsf{ok} \quad i \neq j \supset l_i \neq l_j}{\Theta \vdash [l_1 = \tau_1, \ldots, l_n = \tau_n]\ \mathsf{ok}}$$

$$\frac{\Theta \vdash \tau_1\ \mathsf{ok} \quad \ldots \quad \Theta \vdash \tau_n\ \mathsf{ok} \quad i \neq j \supset l_i \neq l_j}{\Theta \vdash < l_1 = \tau_1, \ldots, l_n = \tau_n >\ \mathsf{ok}}$$

The basic typing judgment is

$$\frac{}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e : \tau}$$

Θ is a set of type variables. Γ is a partial mapping from (unrestricted) variables to types. Δ and Δ_0 are partial mappings from linear variables to types. The judgment says that under Θ, Γ, and Δ, the expression e has type τ but leaves resources Δ_0. An entire program is only well-typed if Δ_0 is empty.

$$\frac{}{\Theta; \Gamma; \Delta \backslash \Delta \vdash s : \texttt{string}} \qquad \frac{}{\Theta; \Gamma; \Delta \backslash \Delta \vdash b : \texttt{bool}}$$

$$\frac{}{\Theta; \Gamma; \Delta, x : \tau \backslash \Delta \vdash x : \tau} \qquad \frac{}{\Theta; \Gamma, x : \tau; \Delta \backslash \Delta \vdash x : \tau}$$

$$\frac{\Theta; \Gamma; \Delta \backslash \Delta_1 \vdash e_1 : \{\} \quad \Theta; \Gamma; \Delta_1 \backslash \Delta_2 \vdash e_2 : \{\} \quad \ldots \quad \Theta; \Gamma; \Delta_{n-1} \backslash \Delta' \vdash e_n : \tau}{\Theta; \Gamma; \Delta \backslash \Delta' \vdash e_1; e_2; \ldots; e_n : \tau}$$

$$\frac{\Theta; \Gamma; \Delta, x : \tau \backslash \Delta_0 \vdash e : \tau' \quad x \notin \Delta_0}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{lfn}\, x : \tau \Rightarrow e : \tau \multimap \tau'} \qquad \frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e_1 : \tau' \multimap \tau \quad \Theta; \Gamma; \Delta_0 \backslash \Delta' \vdash e_2 : \tau'}{\Theta; \Gamma; \Delta \backslash \Delta' \vdash e_1 \hat{\ } e_2 : \tau}$$

$$\frac{\Theta; \Gamma, f : \tau \multimap \tau'; x : \tau \backslash \emptyset \vdash e : \tau' \quad \tau \ \mathsf{ok} \quad \tau' \ \mathsf{ok}}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{lfix}\, f(x : \tau) : \tau' \,\texttt{is}\, e : \tau \multimap \tau'}$$

$$\frac{\Theta; \Gamma, f : \tau \to \tau', x : \tau; \emptyset \backslash \emptyset \vdash e : \tau' \quad \tau \ \mathsf{ok} \quad \tau' \ \mathsf{ok}}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{fix}\, f(x : \tau) : \tau' \,\texttt{is}\, e : \tau \to \tau'}$$

$$\frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e_1 : \tau' \to \tau \quad \Theta; \Gamma; \emptyset \backslash \emptyset \vdash e_2 : \tau'}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e_1 \ e_2 : \tau}$$

$$\frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e_c : \texttt{bool} \quad \Theta; \Gamma; \Delta_0 \backslash \Delta' \vdash e_t : \tau \quad \Theta; \Gamma; \Delta_0 \backslash \Delta' \vdash e_f : \tau}{\Theta; \Gamma; \Delta \backslash \Delta' \vdash \texttt{if}\, e_c \,\texttt{then}\, e_t \,\texttt{else}\, e_f : \tau}$$

$$\frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e : \texttt{string}}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{print}\, e : \{\}} \qquad \frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e : \{\} \multimap \{\}}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{spawn}\, e : \{\}}$$

$$\frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e : \texttt{schan}\ \tau}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{close}\ e : \{\}} \qquad \frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e : \texttt{rchan}\ \tau}{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash \texttt{close}\ e : \{\}}$$

$$\frac{\tau \ \mathsf{ok}}{\Theta; \Gamma; \Delta \backslash \Delta \vdash \texttt{chan}\ \tau : \{s : \texttt{schan}\, \tau, r : \texttt{rchan}\, \tau\}}$$

$$\frac{\Theta; \Gamma; \Delta \backslash \Delta_0 \vdash e_1 : \texttt{schan}\, \tau \quad \Theta; \Gamma; \Delta_0 \backslash \Delta' \vdash e_2 : \tau}{\Theta; \Gamma; \Delta \backslash \Delta' \vdash \texttt{send}(e_1, e_2) : \texttt{schan}\, \tau}$$

$$\frac{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e : \texttt{rchan}\,\tau}{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash \texttt{recv}\,e : \{channel : \texttt{rchan}\,\tau, data : \tau\}}$$

$$\frac{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e_1 :!\tau' \quad \Theta;\Gamma, x:\tau';\Delta_0\backslash\Delta' \vdash e_2 : \tau}{\Theta;\Gamma;\Delta\backslash\Delta' \vdash \texttt{use}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

$$\frac{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e :< \ldots, f : \tau, \ldots >}{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash \#f\ e : \tau}$$

$$\frac{\Theta;\Gamma;\Delta\backslash\Delta_1 \vdash e_1 : \tau_1 \quad \ldots \quad \Theta;\Gamma;\Delta_{n-1}\backslash\Delta' \vdash e_n : \tau_n}{\Theta;\Gamma;\Delta\backslash\Delta' \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\frac{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e_1 : \tau_1 \quad \ldots \quad \Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e_n : \tau_n}{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash < l_1 = e_1, \ldots, l_n = e_n >:< l_1 : \tau_1, \ldots, l_n : \tau_n >}$$

$$\frac{\Theta;\Gamma;\emptyset\backslash\emptyset \vdash e : \tau}{\Theta;\Gamma;\Delta\backslash\Delta \vdash \texttt{exp}\ e :!\tau} \qquad \frac{\tau\ \mathsf{ok} \quad \tau = [\ldots s : \tau' \ldots] \quad \Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e : \tau'}{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash \texttt{inj}(s, e, \tau) : \tau}$$

$$\frac{\begin{array}{l}\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e : [s_1 : \tau_1, \ldots, s_n : \tau_n] \\ \Theta;\Gamma;\Delta_0, v : \tau_1\backslash\Delta' \vdash e_1 : \tau \quad \ldots \quad \Theta;\Gamma;\Delta_0, v : \tau_n\backslash\Delta' \vdash e_n : \tau \quad v \notin \Delta'\end{array}}{\Theta;\Gamma;\Delta\backslash\Delta' \vdash \texttt{case}_v\ e\ \texttt{of} s_1 \Rightarrow e_1 | \ldots | s_n \Rightarrow e_n : \tau}$$

$$\frac{\tau\ \mathsf{ok} \quad \Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e : \forall t.\tau'}{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e[\tau] : [\tau/t]\tau'} \qquad \frac{\Theta, t;\Gamma;\Delta\backslash\Delta_0 \vdash e : \tau}{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash \texttt{tfn}\ t.e : \forall t.\tau}$$

$$\frac{\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e_1 : \tau' \quad \Theta;\Gamma;\Delta_0, x : \tau'\backslash\Delta' \vdash e_2 : \tau \quad x \notin \Delta'}{\Theta;\Gamma;\Delta\backslash\Delta' \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

$$\frac{\begin{array}{l}\Theta;\Gamma;\Delta\backslash\Delta_0 \vdash e_1 : \{s_1 : \tau_1, \ldots, s_n : \tau_n\} \\ \Theta;\Gamma;\Delta_0, x_1 : \tau_1, \ldots, x_n : \tau_n\backslash\Delta' \vdash e_2 : \tau \quad x_1 \notin \Delta' \quad \ldots \quad x_n \notin \Delta'\end{array}}{\Theta;\Gamma;\Delta\backslash\Delta' \vdash \texttt{lett}\ \{s_1 = x_1, \ldots, s_n = x_n\} = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

4 Dynamic Semantics

The dynamic semantics are very straightforward. First, we define what a value is:

$$\frac{}{s\ \mathsf{value}} \qquad \frac{}{b\ \mathsf{value}} \qquad \frac{}{\mathsf{exp}\ e\ \mathsf{value}}$$

$$\frac{e\ \mathsf{value}}{\texttt{inj}(s, e, t)\ \mathsf{value}} \qquad \frac{}{\texttt{SC}\ l\ \mathsf{value}} \qquad \frac{}{\texttt{RC}\ l\ \mathsf{value}}$$

$$\frac{}{\texttt{lfn}\ x:\tau \rightarrow e\ \textsf{value}} \quad \frac{}{\texttt{lfix}\ f(x:\tau):\tau'\ \texttt{is}\ e\ \textsf{value}} \quad \frac{}{\texttt{fix}\ f(x:\tau):\tau'\ \texttt{is}\ e\ \textsf{value}}$$

$$\frac{}{\texttt{tfn}\ t.e\ \textsf{value}} \quad \frac{e_1\ \textsf{value} \quad \ldots \quad e_n\ \textsf{value}}{\{l_1 = e_1, \ldots, l_n = e_n\}\ \textsf{value}} \quad \frac{}{< l_1 = e_1, \ldots, l_n = e_n >\ \textsf{value}}$$

A process is an evaluation context (a stack of frames) and an expression. An evaluation context is a continuation; it has a hole ($\square$) to be filled in with the result of evaluating the expression. A program is a collection of processes running in parallel along with a state. We define a stepping relation on individual processes $(frame, exp, state) \rightsquigarrow (frame, exp, state)$. This relation also includes the global state (a list of channel locations and their status) because some evaluation steps modify it.

$$\begin{array}{rcl}
(\bullet, v, st) & \rightsquigarrow & \emptyset \\
(\{s_1 = v_1, \ldots, s = \square\} \lhd fr, v, st) & \rightsquigarrow & (fr, \{s_1 = v_1, \ldots, s = v\}, st) \\
(\{s_1 = v_1, \ldots, s = \square, s_m = e_m, \ldots\} \lhd fr, v, st) & \rightsquigarrow & \begin{array}{l}(\{s_1 = v_1, \ldots, \\ s = v, s_m = \square, \ldots\} \lhd fr, e_m, st)\end{array} \\
(\texttt{send}(\square, e) \lhd fr, v, st) & \rightsquigarrow & (\texttt{send}(v, \square) \lhd fr, e, st) \\
(\texttt{close} \lhd fr, \texttt{SC/RC}\ l, (st, l\ \texttt{Open})) & \rightsquigarrow & (fr, \{\}, (st, l\ \texttt{Closed})) \\
(\texttt{close} \lhd fr, \texttt{SC/RC}\ l, (st, l\ \texttt{Closed})) & \rightsquigarrow & (fr, \{\}, st) \\
\begin{array}{r}(\texttt{lett}\ \{s_1 = x_1, \ldots, s_n = x_n\} = \square\ \texttt{in}\ e\ \texttt{end} \lhd fr, \\ \{s_1 = v_1, \ldots, s_n = v_n\}, st)\end{array} & \rightsquigarrow & (fr, [v_1 \ldots v_n / x_1 \ldots x_n]e, st) \\
(\texttt{let}\ s = \square\ \texttt{in}\ e\ \texttt{end} \lhd fr, v, st) & \rightsquigarrow & (fr, [v/s]e, st) \\
(\texttt{print} \lhd fr, s, st) & \rightsquigarrow & (fr, \{\}, st) \\
(\texttt{use}\ s = \square\ \texttt{in}\ e'\ \texttt{end} \lhd fr, \texttt{exp}\ e, st) & \rightsquigarrow & (fr, [e/s]e', st) \\
(\square\, e, \texttt{fix}\ f(x)\ \texttt{is}\ e', st) & \rightsquigarrow & (fr, [\texttt{fix}\ f(x)\ \texttt{is}\ e', e/f, x]e', st) \\
(\square\hat{}e, v, st) & \rightsquigarrow & (v\hat{}\square \lhd fr, e, st) \\
(\texttt{lfix}\ f(x)\ \texttt{is}\ e\hat{}\square \lhd fr, v, st) & \rightsquigarrow & (fr, [\texttt{lfix}\ f(x)\ \texttt{is}\ e, v/f, x]e, st) \\
(\texttt{lfn}\ x \Rightarrow e\hat{}\square \lhd fr, v, st) & \rightsquigarrow & (fr, [v/x]e, st) \\
(\square[\tau] \lhd fr, \texttt{tfn}\ t.e, st) & \rightsquigarrow & (fr, [\tau/t]e, st) \\
(\texttt{if}\ \square\ \texttt{then}\ e_t\ \texttt{else}\ e_f \lhd fr, \texttt{true}, st) & \rightsquigarrow & (fr, e_t, st) \\
(\texttt{if}\ \square\ \texttt{then}\ e_t\ \texttt{else}\ e_f \lhd fr, \texttt{false}, st) & \rightsquigarrow & (fr, e_f, st) \\
(\texttt{inj}(s, \square, t) \lhd fr, v, st) & \rightsquigarrow & (fr, \texttt{inj}(s, v, t), st) \\
(\#f\square \lhd fr, < \ldots, f = e, \ldots >, st) & \rightsquigarrow & (fr, e, st) \\
(\texttt{case}_x\ \square\ \texttt{of} \ldots s \Rightarrow e \ldots \lhd fr, \texttt{inj}(s, v, t), st) & \rightsquigarrow & (fr, [v/x]e, st) \\
(\square; e \lhd fr, \{\}, st) & \rightsquigarrow & (fr, e, st) \\
(\square; e_1; \ldots \lhd fr, \{\}, st) & \rightsquigarrow & (\square; \ldots \lhd fr, e_1, st)
\end{array}$$

$$
\begin{array}{rcl}
(fr, \texttt{let}\ s = e_1\ \texttt{in}\, e_2\ \texttt{end}, st) & \rightsquigarrow & (\texttt{let}\ s = \Box\ \texttt{in}\, e\ \texttt{end} \lhd fr, e_1, st) \\
(fr, e_1; \ldots, st) & \rightsquigarrow & (\Box; \ldots \lhd fr, e_1, st) \\
(fr, \texttt{print}\ e, st) & \rightsquigarrow & (\texttt{print}\Box \lhd fr, e, st) \\
(fr, e_1 e_2, st) & \rightsquigarrow & (\Box\, e_2 \lhd fr, e_1, st) \\
(fr, e_1\hat{\ }e_2, st) & \rightsquigarrow & (\Box\hat{\ }e_2 \lhd fr, e_1, st) \\
(fr, e[\tau], st) & \rightsquigarrow & (\Box[\tau] \lhd fr, e, st) \\
(fr, \texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_f, st) & \rightsquigarrow & (\texttt{if}\,\Box\,\texttt{then}\, e_t\,\texttt{else}\, e_f \lhd fr, e_c, st) \\
(fr, \{s = e, \ldots\}, st) & \rightsquigarrow & (\{s = \Box, \ldots\}, st) \\
(fr, \texttt{inj}(s, e, t), st) & \rightsquigarrow & (\texttt{inj}(s, \Box, t), e, st) \\
(fr, \#f\ e, st) & \rightsquigarrow & (\#f\Box, e, st) \\
(fr, \texttt{lett}\,\{...\} = e\ \texttt{in}\ e'\ \texttt{end}, st) & \rightsquigarrow & (\texttt{lett}\,\{...\} = \Box\ \texttt{in}\ e'\ \texttt{end}, e, st) \\
(fr, \texttt{case}_x\, e\,\texttt{of}\ldots, st) & \rightsquigarrow & (\texttt{case}_x\,\Box\,\texttt{of}\ldots \lhd fr, e, st) \\
(fr, \texttt{send}(e_1, e_2), st) & \rightsquigarrow & (\texttt{send}(\Box, e_2) \lhd fr, e_1, st) \\
(fr, \texttt{recv}e, st) & \rightsquigarrow & (\texttt{recv}\Box, e, st) \\
(fr, \texttt{use}\ s = e_1\ \texttt{in}\, e_2\ \texttt{end}, st) & \rightsquigarrow & (\texttt{use}\ s = \Box\ \texttt{in}\, e_2\ \texttt{end}, e_1, st) \\
(fr, \texttt{chan}\ \tau, st) & \rightsquigarrow & (fr, \{s = \texttt{SC}\ l, r = \texttt{RC}\ l\}, (l\ \texttt{Open}, st))\ \text{(1 new)} \\
(fr, \texttt{close}\ e, st) & \rightsquigarrow & (\texttt{close}\Box \lhd fr, e, st) \\
(fr, \texttt{spawn}\ e, st) & \rightsquigarrow & (\texttt{spawn}\Box \lhd fr, e, st)
\end{array}
$$

Now, we define a transition relation $\gg$ on programs.

$$\frac{(fr, e, st) \rightsquigarrow (fr', e', st')}{(\ldots \| (fr, e) \| \ldots, st) \gg (\ldots \| (fr', e') \| \ldots, st')} \tag{1}$$

$$\frac{}{(\ldots \| (\texttt{spawn}\,\Box \lhd fr, v) \| \ldots, st) \gg (\ldots \| (fr, \{\}) \| (v\hat{\ }\Box \lhd \bullet, \{\}) \| \ldots, st)} \tag{2}$$

$$\frac{(l'\ new)}{\begin{array}{c}(\ldots \| (\texttt{send}(\texttt{SC}\ l, \Box) \lhd fr, v) \| \ldots \\ \ldots \| (\texttt{recv}\ \Box \lhd fr', \texttt{RC}\ l) \| \ldots, (st, l\ \texttt{Open}))\end{array} \gg \begin{array}{c}(\ldots \| (fr, \texttt{SC}\ l') \| \ldots \ldots \| (fr', \{data = v, \\ channel = \texttt{RC}\ l'\}) \| \ldots, (st, l'\ \texttt{Open}))\end{array}} \tag{3}$$

$$\frac{}{(\ldots \| (\texttt{send}(\texttt{SC}\ l, \Box) \lhd fr, v) \| \ldots, (st, l\ \texttt{Closed})) \gg (\ldots \| \emptyset \| \ldots, st)} \tag{4}$$

$$\frac{}{(\ldots \| (\texttt{recv}\,\Box \lhd fr, \texttt{RC}\ l) \| \ldots, (st, l\ \texttt{Closed})) \gg (\ldots \| \emptyset \| \ldots, st)} \tag{5}$$

$$\frac{}{(\ldots \| \emptyset \| \ldots, st) \gg (\ldots \| \ldots, st)} \tag{6}$$

The first rule allows a $\rightsquigarrow$ transition on any process in the program. [2]

Spawn (2) simply creates a new process in the program, applying the function to the empty record.

[2] The scheduler is meant to be "fair" in the usual sense, but this is difficult to capture in a rule without specifying the scheduling policy directly. The LinML implementation included has a fair scheduler.

If two processes are ready to synchronize on a send, then the transaction can proceed. (3) Note that the channel location is removed from the state; no other process has access to this channel because the send and receive halves are linear. (A clever implementation might allocate l' to be equal to l!)

If one end of the channel has been closed (4, 5), then a process trying to receive from or send to it will be blocked forever. This process can be collected. (6) Note however that the semantics do not attempt to detect mutually-blocked threads (deadlock), as it seems pointless to optimize for what is usually considered a mistake. It should be straightforward to adapt the techniques from the next section in order to avoid deadlock altogether.

5 Specifying Protocols

In this section we'll see how to code up simple transition systems in linear types in order to enforce a protocol.

Suppose that you are writing some LinML code for a large non-deterministic string printing program. You fear that a particular co-worker who can't follow directions always violates the pre-arranged protocol for communicating between his module and yours. He is always reading and writing in the wrong order and sharing channels with friends. You want to set up the program in such a way that his module won't even compile if it doesn't communicate with yours properly.

LinML has rather primitive modularity features; your module is a function that takes his module as an argument. It will then instantiate type arguments and pass it the operations it needs to run.

Let's start with a very simple protocol:

1. Client sends a string to the Server (a password)
2. Client receives a bool from the Server (was the password correct?)
3. Done.

Your code will require that the client code have the following type:

```
∀ key1, key2, key3, done . {c0 : key1,
                            c1 : ! ({1 : string, 2 : key1} -o key2),
                            c2 : ! (key2 -o {1 : bool, 2 : key3}),
                            c3 : ! (key3 -o done)}
                             -o done
```

The types $\mathtt{key}_n$ are abstract to the client. They represent the steps of the protocol as numbered above. The key power that linear types give us is the ability to ensure that the client posesses only one token of a $\texttt{key}_n$ type at a time! Therefore, if it is type-correct then it follows the protocol described.

The server implements the argument record $\{c0, ...\}$ as follows. All of the `key` type variables are instantiated at the same type: `{str : schan string, boo : rchan bool}`. Thus the functions `c1`, `c2`, `c3`, have access to a channel for sending strings and one for receiving bools. `c0` gets these from the environment (the server process has the other ends of the channels). Here's the code for `c1`:

```
exp (lfn q : {1 : string, 2 : {str : schan string, boo : rchan bool}} =>
     lett {1=ss, 2=k} = q
     in lett {str=str, boo=boo} = k
        in let newstr = send (str, ss)
           in {str=newstr, boo=boo}
           end
        end
     end)
```

Note that `send`ing produces a new channel, which is packaged up as part of the new key. `c2` is similar, and `c3` simply closes both channels. A fully verbose implementation of this example is included in the source code (`ex.sml`).

Now, for this simple example, we could alternately provide a linear argument of type `string -o bool`, which is much more tasteful. (In fact, it is easy to code up a function of that type from the record provided above.) Consider a new protocol:

1.	Client sends a string to the Server	(version identifier)
2.	Client sends a string to the Server	(query)
3.	Client receives a bool from the Server	(response)
4.	Client sends a string to the Server	(quit message)
	OR	
	Client repeats from step 2	
5.	Done.	

This doesn't have a straightforward representation as a function, but is easily represented using our transition system:

```
∀ key1, key2, key3,
  key4, done . {c0 : key1,
                c1 : ! ({1 : string, 2 : key1} -o key2),
                c2 : ! ({1 : string, 2 : key2} -o key3),
                c3 : ! (key3 -o {1 : bool, 2 : key4}),
                c4 : ! (key4 -o key2),
                c5 : ! (key4 -o done)}
                  -o done
```

It is equivalent, but a little more linear-logicesque to use lazy records:

```
∀ key1, key2, key3,
  key4, done . {c0 : key1,
                c1 : ! ({1 : string, 2 : key1} -o key2),
                c2 : ! ({1 : string, 2 : key2} -o key3),
                c3 : ! (key3 -o {1 : bool, 2 : <k2 : key2, k4 : key4>}),
                c4 : ! (key4 -o done)}
                  -o done
```

It's possible for the server to make choices as well. Consider this protocol:

1.	Client sends a string to the Server	(version identifier)
2.	Server sends a boolean to the client	(to step 4)
	OR	
	Server sends a string to the client	(to step 3)
3.	Client sends a bool to the server	
4.	Server sends a string to the client.	
5.	Done.	

```
∀ key1, key2, key3, key4
  key5, done . {c0 : key1,
                c1 : ! ({1 : string, 2 : key1} -o key2),
                c2 : ! (key2 -o [inl : {1 : string, 2 : key4},
                                 inr : {1 : bool,   2 : key3}]),
                c3 : ! ({1 : bool, 2 : key3} -o key4),
                c4 : ! (key4 -o {1 : string, 2 : key5}),
                c5 : ! (key5 -o done)}
                  -o done
```

6 Conclusions

The behavior that these types describe is a *partial correctness* property: *if* the threads terminate, then they will have followed the protocol. It may be quite devastating still to have non-responsive threads that deadlock themselves or spin in a loop. This kind of property detracts from the power of linear types especially because we can not necessarily rely on our resources being used, even though it is very tempting to think we can. On the other hand, concurrent programs are often meant to run indefinitely; in these situations, partial correctness is quite apt.

The definition of the language requires that unrestricted application is call-by-name. Without this, it would be possible to duplicate channels (`chan` τ is well-typed in the empty linear context). I don't know a better solution to this.

It would be interesting to see if these ideas can be made more convenient or efficient. One problem is that the abstract key types actually carry data; though it seemed to me at first that I should be able to use trivial tokens (like $\{\}$), this was the only way I could get it to work out. Perhaps it would need a more sophisticated module system such as Standard ML's.

6.1 Related Work

- Kobayashi et al. [2] give a linear type system for the Pi Calculus [3]. In their system, channels are also marked with their direction (sending or receiving), and multiplicity (how many times it can be used). Their system is aimed towards a more "foundational" calculus, so they do not include as many connectives.
- Wadler [4] applies a similar treatment to a MinML-like language for the purpose of modelling imperative features like arrays. His type system is somewhat peculiar; my approach is much more closely related to the propositions in linear logic.

References

[1] R. Harper. Programming languages: Theory and practice (draft). 2001.

[2] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[3] R. Milner. Communicating and mobile systems: the pi-calculus, 1999.

[4] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.