COMP3161/COMP9161 Supplementary Lecture Notes

Linear Types

Liam O'Connor

September 21, 2018

Suppose we wanted to model a library inside a functional programming language. We will define a function *borrowBook*, that, given a book's `Title`, returns the `Book` if it's available in the library, and a function *returnBook* that takes in a book and returns nothing of value (just a value of unit type):

$$\begin{array}{lcl} \mathit{borrowBook} & : & \texttt{Title} \to \texttt{Maybe Book} \\ \\ \mathit{returnBook} & : & \texttt{Book} \to \mathbf{1} \end{array}$$

The trouble with this model is that, in a purely functional, we know that if *borrowBook* `"Catch-22"` returns a book $\mathsf{Just}\ b$, then it will *always* return $\mathsf{Just}\ b$. Similarly, if it returns $\mathsf{Nothing}$, then it will *always* return nothing. Therefore, we need to include additional state parameters that indicate the state of the `Library` before and after we borrow or return a book:

$$\begin{array}{lcl} \mathit{borrowBook} & : & \texttt{Title} \to \texttt{Library} \to \texttt{Library} \times \texttt{Maybe Book} \\ \\ \mathit{returnBook} & : & \texttt{Book} \to \texttt{Library} \to \texttt{Library} \end{array}$$

This state passing trick is similar to how we modelled the big-step semantics of **TinyImp** previously.

This approach is still problematic, however, as it allows us to reuse the same library over and over, allowing us to borrow infinite books.

$$\begin{array}{l} \mathit{books}\ \ell = \mathbf{let} \\ \qquad\qquad (\ell', mb) \quad = \quad \mathit{borrowBook}\ \texttt{"Ubik"}\ \ell \\ \qquad\qquad (\ell'', bs) \quad = \quad \mathit{books}\ \ell' \\ \qquad\ \mathbf{in\ case}\ mb\ \mathbf{of} \\ \qquad\qquad \mathsf{Nothing} \to (\ell', \mathsf{Nothing}) \\ \qquad\qquad \mathsf{Just}\ b \to (\ell'', b : bs) \end{array}$$

If we changed the highlighted ℓ' to ℓ, we would be *reusing* the library state from before the book was borrowed, enabling us to borrow the same book infinitely often. Of course, were this a real library, there is only *one* `Library` at a time, and once a book is borrowed from it, the previous `Library` has ceased to exist. We would like to express this reality in our interface.

1 State Monads

Let us define a type `Lib` a which represents some computation that changes the `Library` state and returns a value of type a:

$$\mathbf{type}\ \texttt{Lib}\ a = (\texttt{Library} \to \texttt{Library} \times a)$$

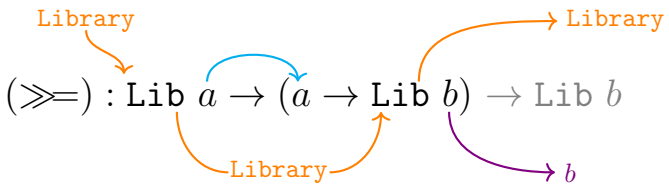
By *hiding* the implementation of this type, and treating it as a kind of black box, we can control how `Lib` computations are composed and manipulated, to enforce that only one `Library` state

should be accessible at one time. Our library interface can be restated as `Lib` computations, with types isomorphic to the original interface:

$$\begin{array}{lcl} \mathit{borrowBook} & : & \texttt{Title} \to \texttt{Lib (Maybe Book)} \\ \mathit{returnBook} & : & \texttt{Book} \to \texttt{Lib } \mathbf{1} \end{array}$$

If, however, the only way to combine multiple `Lib` computations was with a function we define, such as the $(\gg\!=)$ operator, we can ensure that if the $(\gg\!=)$ passes the state linearly through each computation, then any `Lib` computation will only access the `Library` state in a linear way.

The $(\gg\!=)$ operator behaves as follows: Given an input `Library` state, it will run the left hand computation, pass its result to the function on the right-hand side to get the next computation to run, and pass the current `Library` as input to the next computation. Finally, return the result and final `Library` state of that second computation. Diagramatically, it is expressed as follows:



In code, it is something like:

$$(\mathit{la} \gg\!= f) \quad = \quad \lambda \ell.\ \mathbf{let}\ (a, \ell') = \mathit{la}\ \ell\ \mathbf{in}\ \ (f\ a)\ \ell'$$

As can be seen in the diagram, this operation threads the `Library` state linearly through the subcomputations. Combined with the *return* operation, which simply returns the given value without altering the library state, we can make Lib an instance of the Monad abstraction.

$$\begin{array}{l} \mathit{return} :: a \to \mathsf{Lib}\ a \\ \mathit{return}\ x = \lambda \ell.\ (\ell, x) \end{array}$$

Our program that tries to exhaust the library of a particular book now will always return finite lists, as the library is sure to eventually run out of books, and we are unable to "cheat" anymore by re-using stale resources:

$$\begin{array}{l} \mathit{books} :: \texttt{Lib [Book]} \\ \mathit{books} = \mathbf{do} \\ \quad \mathit{mb} \leftarrow \mathit{borrowBook}\ \texttt{"VALIS"} \\ \quad \mathbf{case}\ \mathit{mb}\ \mathbf{of} \\ \qquad \mathsf{Just}\ b \quad \to \quad \mathbf{do}\ \mathit{bs} \leftarrow \mathit{books}; \mathit{return}\ (b : \mathit{bs}) \\ \qquad \mathsf{Nothing} \quad \to \quad \mathit{return}\ [] \end{array}$$

1.1 IO in Haskell

This type of state-passing monad is called a *state monad*. Haskell uses a similar technique to model IO and other side-effects, only instead of a `Library` being the state, we include the *entire world*:

$$\mathbf{type}\ \texttt{IO}\ a = (\text{🌍} \to (\text{🌍}, a))$$

As we don't have a time machine, an IO action like

$$\mathit{launchMissiles} \quad :: \quad \texttt{LaunchCodes} \to \texttt{IO ()}$$

is irrevocable. After launching missiles, we cannot un-launch them, and return to the world before they are launched. This is the same as our `Library` example earlier, where each change to the `Library` could not be simply reverted by returning to a previous state.

1.2 Frame Problems

The more state we add into the hidden state of the monad, the less our type system can help us reason about it. When the state is the entire world, as in:

$$foo :: \texttt{IO}\ ()$$

This `foo` function could literally *do anything*. In general, to accurately specify a stateful computation, we must specify:

1. What changes it makes to the state, and
2. What parts of the state remain unchanged.

As we enlarge the state, the second category grows larger and larger, becoming cumbersome to reason about and difficult to formalise. This problem is called the *frame problem*.

State monads tend to suffer from this problem, as they do not enforce any kind of *separation* between state components.

2 Linear Logic

As we know that lambda-calculus and logic correspond, perhaps a logician has discovered a solution to these problems that does not lead to frame problems but still ensures linear threading of state components.

In this case, that logician was Jean-Yves Girard, and it was the basis of his work in *Linear Logic*. Linear logic is a very deep field of logic and of computer science, however we will just give a cursory explanation for this course, to illustrate the origin of the *linear types* ideas described in the next section.

2.1 The trouble with Logic

Suppose I will eat any cake that I have. I might state this implication:

$$\text{HaveCake} \Rightarrow \text{EatCake}$$

But, following the rules of propositional logic, I could deduce that $\text{HaveCake} \Rightarrow (\text{HaveCake} \wedge \text{EatCake})$. This isn't correct! Once I have eaten the cake, I should no longer have the cake.

A similar situation would be if Alice and Bob both want to read "Dune" by Frank Herbert, but there is only one copy of "Dune" at the library.

We could state two implications, both seemingly true in isolation:

$$\text{DuneAtLibrary} \Rightarrow \text{BobBorrowsDune}$$

and

$$\text{DuneAtLibrary} \Rightarrow \text{AliceBorrowsDune}$$

From this we could conclude that if DuneAtLibrary then $\text{BobBorrowsDune} \wedge \text{AliceBorrowsDune}$, which should be impossible — only one of them can borrow the book.

There are ways of encoding this more accurately in standard logic, but they usually run into the same frame problem discussed earlier.

We want to say that if an assumption is *used* in a derivation, it should not be used again. This is similar to the linearity constraints we wanted to impose on our `Library` state earlier.

2.2 Substructural Logic

Normally, we treat our *context* of assumptions Γ as a *set*:

$$\frac{A \in \Gamma}{\Gamma \vdash A}\textsc{Assumption}$$

More formally, we are allowing the assumptions to be *duplicated*, *discarded*, or *rearranged* using the following *structural rules*, and when using an assumption we require that it is the *only* assumption in the context:

$$\frac{}{P \vdash P}\textsc{Assumption}$$

$$\frac{\Gamma_2, \Gamma_1 \vdash P}{\Gamma_1, \Gamma_2 \vdash P}\textsc{Exchange} \qquad \frac{Q, Q, \Gamma \vdash P}{Q, \Gamma \vdash P}\textsc{Contraction} \qquad \frac{\Gamma \vdash P}{Q, \Gamma \vdash P}\textsc{Weakening}$$

These rules allow us to treat our list of assumptions Γ as though it was a set. After all, a list where you can rearrange elements, duplicate elements, or drop redundant elements at-will is essentially a set.

Linear Logic removes the two rules of Contraction and Weakening, turning our context into a *multiset*. Changing the order of assumptions (Exchange) is still allowed. This means that every derivation must use each assumption *exactly once*. Such a logic is called *substructural*, as it has restricted or removed the *structural* rules.

In Linear Logic, you can view logical atoms as *resources*. There are a number of connectives in Linear Logic, more than are in this table here, but this should be sufficient to give you an idea of how it works, without delving into too much formal detail:

$A \multimap B$	A can be transformed into B.
$A \otimes B$	You've got both A and B.
$A \oplus B$	You've got either A or B, you don't get to choose.
$A \,\&\, B$	You can pick from A or B.
$!A$	You've got an unlimited amount of A.

For example, our original cake scenario could be described as HaveCake $\multimap$ EatCake, which indicates that the HaveCake assumption is *used up* by the derivation of EatCake. Thus we can't make the same erroneous conclusion that we could in traditional logic.

We could also restate our problematic library scenario as:

$$\text{DuneAtLibrary} \multimap \text{BobBorrowsDune}$$

$$\text{DuneAtLibrary} \multimap \text{AliceBorrowsDune}$$

This way, if we know DuneAtLibrary then we only know that BobBorrowsDune$\oplus$AliceBorrowsDune – only one of them may borrow the one book that is in the library.

An example of something that uses all of the connectives above would be something like a specification of a lunch menu, as below:

> For \$10, you get one serving of tempura, as much rice as you like, your choice of side salad or miso soup, and the dessert of the day (fruit or ice cream, depending on season).

This specification can be translated into the following logical statement:

$$\begin{gathered}\$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \\ \multimap \\ \text{Tempura} \otimes\, !\text{Rice} \otimes (\text{Salad} \,\&\, \text{Miso}) \otimes (\text{Fruit} \oplus \text{IceCream})\end{gathered}$$

3 Linear Types

To define a linear type system based on the ideas of linear logic, we would expect to end up with a language where each variable in scope must be used *exactly once*. Let us start by first removing the two structural rules of contraction and weakening:

$$\frac{x : \tau, x : \tau, \Gamma \vdash e : \rho}{x : \tau, \Gamma \vdash e : \rho}\textsc{Contraction} \qquad \frac{\Gamma \vdash e : \rho}{x : \tau, \Gamma \vdash P : \rho}\textsc{Weakening}$$

We will also restate our variable typing rule to require that the only variable left in the context is the variable we are typing:

$$\frac{}{x : \tau \vdash x : \tau}\text{Var}$$

As we do not allow assumptions in our context to be duplicated anymore, expressions that involve two or more evaluations must split the context between their subexpressions, as the variables may only be used once:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1\Gamma_2 \vdash (\texttt{App}\ e_1\ e_2) : \tau_2}$$

Note here that the normal context Γ has been split into Γ_1 and Γ_2.

Expressions that branch use the same context in each branch, so that we ensure that the variables are used no matter which branch is taken:

$$\frac{\Gamma_1 \vdash e_1 : \texttt{Bool} \quad \Gamma_2 \vdash e_2 : \tau \quad \Gamma_2 \vdash e_3 : \tau}{\Gamma_1\Gamma_2 \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

This means that *every* variable in scope has to be used *exactly once* in each branch of control flow. Restating our book-borrowing function from earlier, but using linear types, we get:

$$\mathit{borrowBook} :: \texttt{Title} \to \texttt{Library} \to \texttt{Library} \otimes (\texttt{Maybe Book})$$

$$\begin{aligned}
&\mathit{books} :: \texttt{Library} \to \texttt{Library} \otimes [\texttt{Book}] \\
&\mathit{books}\ \ell = \textbf{let} \\
&\quad (\ell', \mathit{mb}) = \mathit{borrowBook}\ \ell\ \texttt{"VALIS"} \\
&\ \textbf{in case}\ \mathit{mb}\ \textbf{of} \\
&\quad \mathsf{Just}\ b \quad \to \quad \textbf{let}\ (\ell'', \mathit{bs}) = \mathit{books}\ \ell'\ \textbf{in}\ (\ell'', (b : \mathit{bs})) \\
&\quad \mathsf{Nothing} \quad \to \quad (\ell', [])
\end{aligned}$$

In this example, changing the `Library` parameter ℓ' (given to *books*) to ℓ would result in two type errors: The first would be because the variable ℓ is used more than once (once in the *borrowBook* application and once in the *books* application), and the second would be because the variable ℓ' now goes unused in that branch of the control flow.

Thus, we have used linear types here to enforce that the state is passed around in a linear way, and not accessed more than once.

3.1 Shareability

The linear type system presented above is not ideal, however, as this makes *all* variables linear, even things that can be very cheaply duplicated or discarded like machine `Int`s. Consider this example:

$$\textbf{let}\ x = 5\ \textbf{in}\ x + x$$

Here the type checker would raise an error as the variable x is used more than once. For this reason, it is desirable to *combine* linear and non-linear types in the one system. There are many

ways to do this, but a simple way is to re-enable contraction and weakening, but only for those types that are deemed safe to **Share**:

$$\frac{x : \tau, x : \tau, \Gamma \vdash e : \rho \quad \tau \ \mathbf{Share}}{x : \tau, \Gamma \vdash e : \rho}\mathrm{C} \qquad \frac{\Gamma \vdash e : \rho \quad \tau \ \mathbf{Share}}{x : \tau, \Gamma \vdash P : \rho}\mathrm{W}$$

The exact definition of **Share** depends on the language designer, but we shall assume basic types like `Int` can be shared, but something like `Library` cannot.

3.2 Uniqueness Types

Linear types can be used to manage resources such as file handles:

$$\mathit{openFile} : \texttt{FileName} \to \texttt{File}$$

$$\mathit{writeFile} : \texttt{String} \to \texttt{File} \to \texttt{File}$$

$$\mathit{closeFile} : \texttt{File} \to \mathbf{1}$$

Making `File` linear ensures that we don't write to stale `File`s, or forget to run `closeFile` on any `File`s.

It even allows us to view the `File` as a purely functional string, and continue reasoning about our program equationally as if it was a purely functional program, even though it is performing file IO! We know this is true because the type system ensures there can only be one active `File` handle for a given file at a given time. Such a type scheme is called *uniqueness* types. We can also make use of uniqueness types for managing memory:

$$\begin{array}{lcl} \mathit{malloc} & : & \texttt{Size} \to \texttt{Buffer} \\ \mathit{poke} & : & \texttt{Offset} \to \texttt{Char} \to \texttt{Buffer} \to \texttt{Buffer} \\ \mathit{free} & : & \texttt{Buffer} \to \mathbf{1} \end{array}$$

In this way, we ensure that each *malloc* has a corresponding *free*, and obviate the need for garbage collection, while still ensuring memory safety. This is similar to the approach used by the Rust programming language, among others.

3.3 Linear Products

Our product types may be constructed in the usual way, although we use the Linear Logic symbol $\otimes$ now rather than $\times$:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1\Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}$$

They can be shared iff their components can be shared:

$$\frac{\tau_1 \ \mathbf{Share} \quad \tau_2 \ \mathbf{Share}}{\tau_1 \otimes \tau_2 \ \mathbf{Share}}$$

However, they cannot be unpacked in the normal way we defined for products earlier, namely fst and snd:

$$\mathsf{fst} : a \otimes b \to a \qquad \mathsf{snd} : a \otimes b \to b$$

This is because of the *other* side of the tuple is a linear value, then we are discarding a linear value without using it. For example, if we apply fst to a pair $\texttt{Int} \otimes \texttt{File}$, we would get back the `Int`, but the pair, being used, would no longer be accessible, thus removing the `File` from scope without calling *closeFile* on it first.

We need a special form of expression to unpack pairs without discarding one of the elements. To do that, we write a special `Split` construct, that, given a pair, binds two variables (one for each componenet) in the scope of a nested expression:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad x : \tau_1, y : \tau_2, \Gamma_2 \vdash e_2 : \tau}{\Gamma_1\Gamma_2 \vdash (\texttt{Split}\ e_1\ (x.y.\ e_2)) : \tau}$$

3.4 Linear Functions

As top-level functions do not typically contain values, one might be inclined to think that they can always be shared, as they will not contain any linear values inside them. When we introduced *closures*, however, we introduced function values that *do* contain values in their captured closure environment. If we allow functions to be freely shared, we can write a function that duplicates linear values by capturing them in a closure:

$$\begin{aligned} &\mathit{dupLibrary} : \texttt{Library} \to \texttt{Library} \otimes \texttt{Library} \\ &\mathit{dupLibrary}\ \ell = \ \textbf{let}\ f = (\lambda\_.\ \ell)\ \textbf{in}\ (f\ (), f\ ()) \end{aligned}$$

To address this problem, we tweak our (shareable) function typing rule to require that any variables that are captured in the function's definition are shareable:

$$\frac{\text{all } \Gamma\ \textbf{Share} \qquad f : \tau_1 \to \tau_2, x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash (\texttt{Fun}\ (f.x.\ e)) : \tau_1 \to \tau_2} \qquad \frac{}{\tau_1 \to \tau_2\ \textbf{Share}}$$

We also add a new type of function, a *linear* function $\tau_1 \multimap \tau_2$ which can only be called once, but allows all types of variables to be captured:

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash (\texttt{LinFun}\ (x.\ e)) : \tau_1 \multimap \tau_2}$$

Naturally these linear functions are not recursive as then they would be used twice: once when the function was called recursively, and once when the function was called from outside.

3.5 Reading Values

Suppose we wanted to write a function that returns the size of a file. A type like

$$\mathit{sizeOfFile} :: \texttt{File} \to \texttt{Int}$$

would not suffice, as this would *destroy* the file while returning its size. Instead, we might think to return the file along with its size like this:

$$\mathit{sizeOfFile} :: \texttt{File} \to (\texttt{Int} \otimes \texttt{File})$$

But this is very heavy-handed and cumbersome, as the type now seems to indicate that the file may be *changed* in some way by *sizeOfFile*.

What we would like to do is write

$$\mathit{sizeOfFile} :: !\texttt{File} \to \texttt{Int}$$

Where !File indicates a file that is *read-only* but freely **Share**able.

We will introduce the expression form **let!** $(v)\ x = e_1$ **in** e_2, which makes a linear variable $v : \rho$ into a *temporarily shareable* version $!\rho$ inside e_1, but keeps the original $v : \rho$ in scope for e_2 . $!\tau$ is called an *observer* in the literature or a *borrow* in Rust.

$$\frac{v : !\rho, \Gamma_1 \vdash e_1 : \tau \qquad v : \rho, x : \tau_1, \Gamma_2 \vdash e_2 : \tau_2}{v : \rho, \Gamma_1\Gamma_2 \vdash \textbf{let!}\ (v)\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

Then, using our revised *sizeOfFile* above, we can call it on a (writable) File f by writing:

$$\textbf{let!}\ (f)\ \mathit{size} = \mathit{sizeOfFile}\ f\ \textbf{in} \ldots$$

3.5.1 Ensuring Purity

If we imagine that the file is destructively updated, then we could use **let!** to have both a `File` and an observer of that file in scope at the same time:

$$\textbf{let!}\ (f)\ x = f\ \textbf{in}\ (\mathit{writeFile}\ \texttt{"Hello"}\ f, x)$$

Changes to f could be observed through x, thus breaking purity and ruining our equational semantics.

To ensure that we can keep a purely functional semantics with **let!**, we must ensure that the read-only value f is not part of the value bound in the **let!**. In general this can be done by *escape analysis*, but This can be addressed by enforcing that the observer cannot be bound in the **let!**, either using *escape analysis* or some type-based approximation, such as ensuring that $!\texttt{tau}_1$ does not occur in the type ρ.

3.6 Applications

Linear and uniqueness types are becoming increasingly popular in many languages:

- The language Cogent (my PhD) uses *linear uniqueness types* to manage effects and state, aimed at simplifying formal verification.
- Rust makes use of *uniqueness types* to ensure memory safety and garbage collection, even though it does not have a functional semantics.
- Haskell and Swift have plans to adopt linear or uniqueness types at various levels of maturity.
- Idris has a basic linear type system extension.