COMP[39]161
Concepts of Programming Languages

Effects and Linear Types

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Motivation

Suppose we wanted to model a library using a purely functional programming language.

$$
\begin{aligned}
\mathit{borrowBook} &: \mathtt{Title} \to \mathtt{Maybe\ Book} \\
\mathit{returnBook} &: \mathtt{Book} \to \mathbf{1}
\end{aligned}
$$

This is not purely functional. Either *borrowBook* `"Ubik"` will always return `Nothing` or the book must always be available

$\Rightarrow$ infinite books 

There is some hidden state here.

State-Passing

We can use a similar trick to the big-step semantics of **TinyImp**, and pass the state (the `Library`) as an input to and output of each function:

$$
\begin{aligned}
\mathit{borrowBook} &: \texttt{Title} \to \texttt{Library} \to \texttt{Library} \times (\texttt{Maybe Book}) \\
\mathit{returnBook} &: \texttt{Book} \to \texttt{Library} \to \texttt{Library}
\end{aligned}
$$

I can still get infinite books here, just by borrowing the books from the same `Library`. Demonstrate on whiteboard

We need to enforce that, after the `Library` state has been passed in to say, *borrowBook*, the same `Library` will never be used again. We also presumably want to enforce that the `Library` will never be destroyed by the garbage collector.

Encapsulation

Let's define a type $\texttt{Lib}\ a$ which represents some computation that changes the $\texttt{Library}$ state and returns a value of type a.

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

Then our library interface can be restated as $\texttt{Lib}$ computations:

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

Observe that these types are isomorphic to the ones on the previous slide. What does this buy us?

Remember that we want the $\texttt{Library}$ state to be threaded linearly through the program.

Composition

If we make `Lib` an *abstract data type* that hides the internal definition, but expose our interface functions, we can control how `Lib` computations are composed by just exposing one composition function:

Library Library

$$(\gg=) : \texttt{Lib}\ a \to (a \to \texttt{Lib}\ b) \to \texttt{Lib}\ b$$

Library b

(implement on whiteboard)

Combine it with a function $\mathit{return} : a \to \texttt{Lib}\ a$ and we have ourselves a *monad*.

The State Monad

Without exposing the implementation of `Lib`, it's now impossible to get infinite books, as we can only borrow as many books as the library has.

```haskell
loop :: Lib [Book]
loop = do
  mb ← borrowBook "VALIS"
  case mb of
    Just b   →  do bs ← loop; return (b : bs)
    Nothing  →  return []
```

This program will always return finite lists, because we can't access the old `Library` after we've already borrowed from it.

IO in Haskell

Haskell uses the same technique to model IO or any imperative computation.

$$\textbf{type}\ \mathtt{IO}\ a = (🌍 \to (🌍, a))$$

$$
\begin{array}{lcl}
\mathit{putStrLn} & :: & \mathtt{String} \to \mathtt{IO}\ () \\
\mathit{getChar} & :: & \mathtt{IO}\ \mathtt{Char} \\
\mathit{launchMissiles} & :: & \mathtt{LaunchCodes} \to \mathtt{IO}\ ()
\end{array}
$$

As we don't have a time machine, these actions are irrevocable. Thus, the same trick of hiding the implementation of `IO` is used to enforce *linearity* of the 🌍.

This approach has some significant drawbacks, however...

Hoare Logic

We can specify our `Lib` computations using Hoare Logic, similar to **TinyImp**.
Modelling the `Library` as a multiset of books L:

$$\{L = L_0\}$$

$$\mathit{borrowBook}\ \texttt{"Ubik"}$$

$$\{L = L_0 \setminus \{\texttt{"Ubik"}\}\}$$

Read in English, this says "If the library is L_0 before I borrow the book, then after I borrow the book, the library will be L_0 except with the book I wanted removed."

State Composition

What if we had multiple libraries?
We could try passing all the libraries as our hidden state (here just two):

$$\textbf{type}\ \texttt{Libs}\ a = ((\texttt{Library} \times \texttt{Library}) \to (\texttt{Library} \times \texttt{Library}) \times a)$$

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

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

Specifying this

Assuming the two libraries are A and B, does our old specification still suffice?

$$\{A = A_0\}$$

$$\mathit{borrowBook}_A\ \texttt{"Ubik"}$$

$$\{A = A_0 \setminus \{\texttt{"Ubik"}\}\}$$

No. We didn't state that the other library did not change:

$$\{A = A_0\ \wedge B = B_0\}$$

$$\mathit{borrowBook}_A\ \texttt{"Ubik"}$$

$$\{A = A_0 \setminus \{\texttt{"Ubik"}\}\ \wedge B = B_0\}$$

This is called the *frame problem*. It's because the type system doesn't tell us which parts of the state are affected.

Non-compositionality

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:

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

This `foo` function could literally do anything.

Looking to Logic

Because logics and programming languages correspond (from the Curry-Howard correspondence), surely some logician has found a solution to these kinds of problems in that domain.

In this case, that logician was Jean-Yves Girard, and it was the basis of his work in *Linear Logic*.

Linear Logic is widely used in computer science, however our treatment of it in this course is cursory.

The Problem in Regular Logic

Suppose Alice and Bob both want to read “Dune” by Frank Herbert. There is only one copy of “Dune” at the library. We could state two implications, both seemingly true in isolation:

$$\mathrm{DuneAtLibrary} \rightarrow \mathrm{BobBorrowsDune}$$

and

$$\mathrm{DuneAtLibrary} \rightarrow \mathrm{AliceBorrowsDune}$$

From this we could conclude that if $\mathrm{DuneAtLibrary}$ then $\mathrm{BobBorrowsDune} \wedge \mathrm{AliceBorrowsDune}$ — which should be impossible.

Standard logic lets us have our cake and eat it too.

Fixing this issue

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

We want to say that if an assumption is used, it should not be used again.

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*:

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

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

Substructural Logics

Linear Logic is a *substructural logic*. It removes some of these rules, namely, *contraction* and *weakening*:

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

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

This means we can't duplicate or discard assumptions anymore, only rearrange them. It makes our context into a multiset.

Linear Connectives

In Linear Logic, you can view logical atoms as resources.

$A \multimap B$	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.

Example (Lunch Special)

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).

$$\$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1 \otimes \$1$$
$$\multimap$$
$$\mathrm{Tempura} \otimes !\mathrm{Rice} \otimes (\mathrm{Salad} \,\&\, \mathrm{Miso}) \otimes (\mathrm{Fruit} \oplus \mathrm{IceCream})$$

Back to the Library

In Linear Logic, we could restate our problematic scenario as:

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

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

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

Back to PLs

We want to backport these ideas to a programming language!
Let's start by eliminating the same structural rules:

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

Expressions that involve two evaluations split the context:

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

Expressions that branch use the same context in each branch:

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

This means that every variable in scope has to be used exactly once in each branch of control flow.

Library Example

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

$$\begin{array}{l}
loop :: \texttt{Library} \to \texttt{Library} \otimes [\texttt{Book}] \\
loop\ \ell = \textbf{let} \\
\boxed{\text{Unused}} \to (\ell', mb) = borrowBook\ \ell\ \texttt{"VALIS"} \\
\quad \textbf{in case}\ mb\ \textbf{of} \\
\qquad \mathsf{Just}\ b \quad \to \quad \textbf{let}\ (\ell'', bs) = loop\ \ell\ \ \textbf{in}\ (\ell'', (b : bs)) \\
\qquad \mathsf{Nothing} \quad \to \quad (\ell', [])
\end{array}$$

$$\boxed{\text{Used more than once}}$$

Linear types prevent us from being able to re-use the `Library` or discard it.

Not Everything is Linear

Some things should not be linear. For example, currently linear types would reject the following program:

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

Clearly, it's okay to use `Int`s multiple times, just not anything like, say, `Library`.

We'll bring back contraction and weakening, but only for types that can be shared, which we'll write as the judgement τ **Share**.

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

Basic types like `Int` can be shared, but something like `Library` cannot.

Managing Resources

Linear types can be used to give purely functional interfaces to mutable, or destructively updated things:

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

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

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

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

Product Types

Our product types may be constructed in the usual way:

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

They can be shared iff their components can be shared:

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

Can they be destructed in the normal way?

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

No, the other side of the pair is discarded without being used!

Splitting Products

We need a special form of expression to unpack pairs without discarding one of the elements:

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

Functions

Can we make functions shareable?

If a function captures a linear variable in its closure, then it is possible to circumvent the linear type system and get multiple copies of the same linear resource:

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

Solution

One shareable function type, $\tau_1 \rightarrow \tau_2$, and a linear function type, $\tau_1 \multimap \tau_2$. The former requires all variables in the captured context to be shareable, but can be itself shared:

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

Linear functions $\tau_1 \multimap \tau_2$ can only be called once:

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

Reading Pains

Suppose we wanted a function to get the size of a file. What type should it have?

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

This is cumbersome. It appears as though getting the size of the file might change the file.Is there some way to get a shareable, read-only version of the File?

Let!

The expression **let!** $(v)\ x = e_1$ **in** e_2 makes a linear variable $v : \rho$ into a temporarily shareable version $!\rho$ inside e_1. $!\rho$ is called an observer in the literature or a borrow in Rust.

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

Then we can have:

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

And call it on a file with:

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

Can we duplicate or discard linear resources with this? Can we violate purity?

Making Let! Safe

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. This can be addressed by enforcing that the observer cannot be bound in the **let!**, either using escape analysis or some type-based approximation.

Uniqueness Types

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

If all heap-allocated objects are linear, then we ensure that there is only one writable pointer to any heap object at a time. This makes *uniqueness types*.

1. The type system ensures all *malloc* calls have a corresponding *free* $\Rightarrow$ No need for garbage collection.
2. The uniqueness of pointers means that *poke* can destructively update the buffer, not create a new one.
3. We can use observers to state in the types that *peek* does not write to the buffer.
4. We have both a stateful update semantics, and a purely functional value semantics.

Applications

- 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.