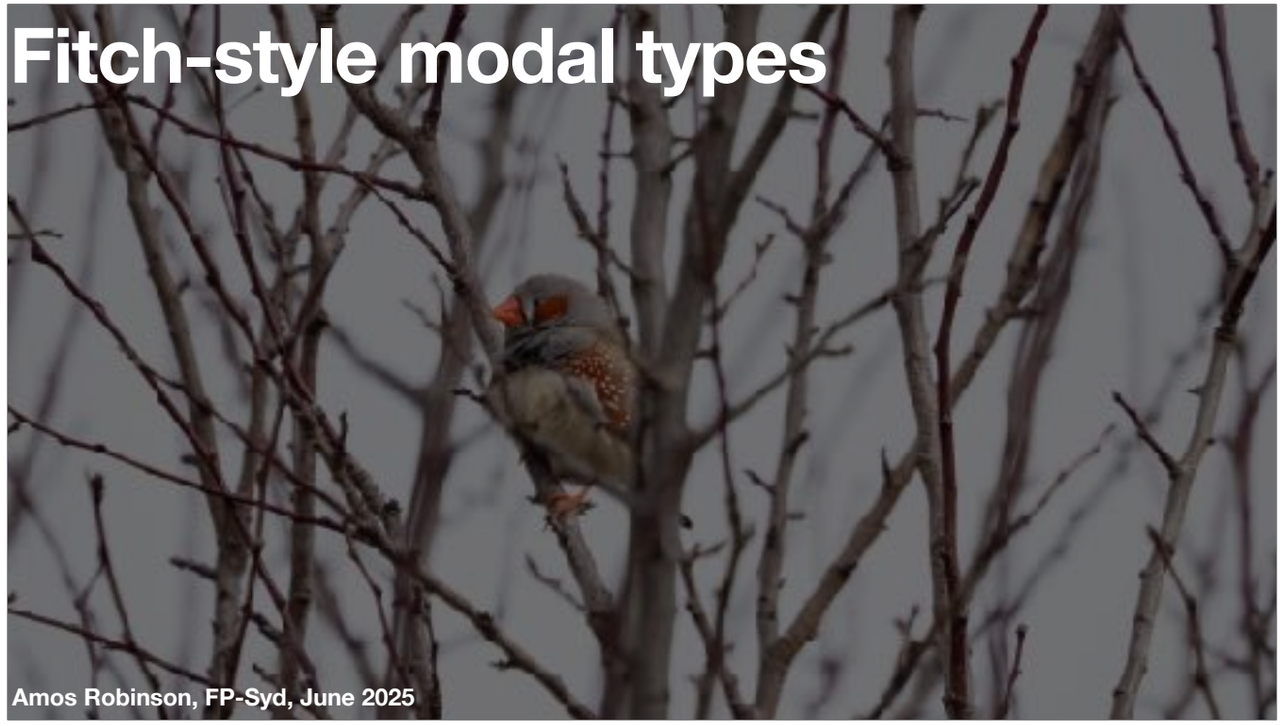
Fitch-style modal types



Amos Robinson, FP-Syd, June 2025

Modal logic - all possible worlds

$\Box(1 \leq \text{DiceRoll} \leq 6)$

In all worlds, our dice roll is between 1 and 6

$\text{DiceRoll} = 6$

In this world, the dice roll results in 6

$\Box(\text{DiceRoll} = 6 \implies \text{Win})$

In all worlds, if the dice roll is 6, then we win

Win

In this world, we won

Modal types come from modal logic.
Modal logic extends propositional logic with "modalities", which are operators that let us interpret propositions in different contexts.
One of the most common modal logics is "necessity" or "all possible worlds".
If we're playing a dice game, we can interpret the "all possible worlds" to mean all of the possible instances of the game.

Applications of modal types

- Template metaprogramming / staged compilation
  - $\Box A$: compile-time code that produces a runtime value of type A
- Streaming
  - $\Box A$: a stream of values of type A
- Purity in an impure language:
  - $\Box A$: a pure computation of type A with no external dependencies

Modal types bring this idea of "modalities" to programming languages.
A "modality" lets us interpret types in different "contexts".
There are a bunch of different applications for this, but one of the most well-studied applications is for template metaprogramming, like in typed Template Haskell or MetaOCaml.
In this kind of template metaprogramming, a type "box A" means a compile-time expression that will produce a runtime value of type A.

Streaming is another application, where type "box A" means "a stream of As".

For impure languages like OCaml or Scala, modal types can be used to implement pure computations, where "box A" means a pure computation that can't depend on the values of mutable variables.
This idea of pure computation is similar to the "all possible worlds" interpretation from modal logic: the computation "box A" produces the same value in all possible mutable heaps (worlds).

Common axioms for modal types

K: $\Box (A \to B) \to \Box A \to \Box B$

R: $A \to \Box A$

T: $\Box A \to A$

In modal logic, the different logic variants are characterised by which axioms they support.
These axioms are very familiar from functional programming.

The most important axiom is "K", which is analogous to the "lifted application" form of applicative functors. It takes a boxed function "A to B" and a boxed argument "A" and applies the function to get a boxed "B". As far as I'm aware, all interesting modal logics and modal types support this axiom.

The next axiom is return, which takes a value "A" and boxes it up. This is supported by some modal logics, but not all of them: our "all possible worlds" modal logic doesn't support this, because it says "anything true in this world is true in all worlds".

The "T" or eval/extract axiom is the dual of return, and talks about extracting a value "A" from a boxed value. Our "all possible worlds" logic does support this.

There are more complex axioms too, but these are the most important ones.

Staging: axiom K

add : exp int → exp int → exp int

add x y = box (unbox x + unbox y)

Here's an example where we implicitly use axiom "K" in a staged compilation context in a Fitch-style type system.
Our add function takes two boxed ints. We unbox both, add them together, and then box it back up.
Later, we'll see the restrictions on the unbox operator, which only let us unbox values if the result is boxed up again.

Staging: axiom K

```
k : exp (a → b) → exp a → exp b

add (x: exp int) (y: exp int) =
  let add__ : exp (int → (int → int)) = box (+) in
  let addx_ : exp (int → int)         = k add__ x in
  let addxy : exp int                 = k addx_ y in
  addxy
```

We could also use the axiom K explicitly. It's messy here, but it's not too bad if we clean it up with some applicative functor syntax.

Staging: no R (return)

```
return : a → exp a

staged_query (): exp (list user) =
  let db = Db.connect "127.0.0.1" in  (* connection opened at compile-time *)
  box (Db.read_table db)                  (* …but used at runtime! *)
```

Offline metaprogramming (eg Template Haskell) lets us generate some code at compile-time, and have it execute later at runtime.
Since we might be running on a different machine to where we compiled, we can't serialise things like file handles and sockets.
That means that we can't just refer to compile-time values and expect them to be serialised to our program on disk.
We're implicitly trying to use the return axiom here, but this is a type error.

Online / runtime staging like MetaOCaml can support this case!

Staging: no T (eval)

```
eval : exp a → a

staged_args (): list string =
  let read_args = box (System.getArgs ()) in  (* read runtime arguments... *)
  eval read_args                              (* at compile time *)
```

Similarly, we can't, at compile-time, read from variables that won't exist until runtime.
If we read the command-line arguments of the runtime program, and try to evaluate it at compile-time, we might be able to read the compiler's command-line arguments, but it's not clear if that's meaningful.
This program explicitly uses the eval function, so it's easy to rule out by not implementing eval.

Online / runtime staging like MetaOCaml can support this case!

Common axioms for modal types - matrix

	Staging	Streaming	Purity
K: $\Box(A \to B) \to \Box A \to \Box B$	✅	✅	✅
R: $A \to \Box A$	❌	✅	❌
T: $\Box A \to A$	❌	❌	✅

Our different applications of modal types support the different axioms.

Our particular staging library doesn't support R or T.

We could imagine implementing R in a streaming context by just replicating the value over the stream.
But it doesn't really make sense to implement extracT for streams, because we can't get a value from an empty stream.

For purity, we can't implement R because, if the ambient calculus is impure, the input value has been computed impurely. We *can* implement extracT, because we can use a pure value in an impure context.

We have three different kinds of "computational contexts" here that support different operations. Streaming is an applicative functor (and might be a monad too, depending on your requirements), but neither of the others are. The cool thing about modal types is that they give us a way to unify these different modes of computation.

Staging - type system

So, let's look at how a Fitch style modal type system actually works. We'll focus on the application of offline template metaprogramming.

Staging - grammar

$$e ::= f \mid x \mid \lambda x.\, e \mid e\, e \mid \text{box } e \mid \text{unbox } e$$

$$\tau ::= \text{int} \mid \text{real} \mid \cdots \mid \Box\tau$$

$$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \text{Lock}$$

Our expression grammar is pretty normal with references to const functions (eg add and Db.read_table), variables, lambdas and applications.
We also have box and unbox operators.

Types get a new box tau type representing expressions.

The most interesting addition is the typing context: we add a "lock" that locks up the variables to the left of the context.
We use this to control access to variables when we're inside a box, but we'll see this in action in the typing rules.

Staging - typing rules

$$\frac{\Gamma, \text{Lock} \vdash e : \tau}{\Gamma \vdash \text{box } e : \square \tau}(\text{Box}) \qquad \frac{\Gamma \vdash e : \square \tau \qquad \text{Lock} \notin \Gamma'}{\Gamma, \text{Lock}, \Gamma' \vdash \text{unbox } e : \tau}(\text{Unbox})$$

$$\frac{\text{Lock} \notin \Gamma'}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}(\text{Var}) \qquad \frac{f : \tau \in \text{Defs}}{\Gamma \vdash f : \tau}(\text{Const})$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau \,.\, e) : \tau \to \tau'}(\text{Abs}) \qquad \frac{\Gamma \vdash e : (\tau' \to \tau) \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}(\text{App})$$

The white typing rules are pretty standard; the green ones are specific to our system.

The box rule says that if we lock up the context and typecheck the expression, then we can box up the result type.

Locking the context has two main purposes, which we can see in how the locked context interacts with the Var rule and the Unbox rule.

In the Var rule, the lock tells us we're inside a "box" expression, and here the lock stops us from accessing variables outside of the boxed context. The condition "Lock is not in Gamma'" means that we can't have a lock between where the variable is declared in the context and where the variable is used in the expression.

The unbox rule is almost the dual of the box rule. Box lets us trade a locked context with an unboxed value for a boxed result; unbox lets us trade a boxed result for a locked context with an unboxed value.

Unbox and box are almost duals, except for the Gamma' in unbox. The Gamma' lets us throw away some of the context, as long as it doesn't contain any locks. Forgetting variables or "weakening" is pretty common operation in lambda calculi, and we just need to make sure we don't throw away any locks. If we wanted to, we could split this out to a separate weakening rule and they'd be dual.

Staging - boxed addition

$$\frac{\Gamma, \text{Lock} \vdash e : \tau}{\Gamma \vdash \text{box}\ e : \square\tau}\text{(Box)} \qquad \frac{\Gamma \vdash e : \square\tau \qquad \text{Lock} \notin \Gamma'}{\Gamma, \text{Lock}, \Gamma' \vdash \text{unbox}\ e : \tau}\text{(Unbox)}$$

$$\frac{\text{Lock} \notin \Gamma'}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}\text{(Var)}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{}{x : \square\,\text{int}, y : \square\,\text{int} \vdash x : \square\,\text{int}}\text{(Var)}}{x : \square\,\text{int}, y : \square\,\text{int}, \text{Lock} \vdash \text{unbox}\ x : \text{int}}\text{(Unbox)} \quad \cdots}{x : \square\,\text{int}, y : \square\,\text{int}, \text{Lock} \vdash (\text{unbox}\ x + \text{unbox}\ y) : \text{int}}\text{(App)}}{x : \square\,\text{int}, y : \square\,\text{int} \vdash \text{box}\ (\text{unbox}\ x + \text{unbox}\ y) : \square\,\text{int}}\text{(Box)}$$

We can typecheck the boxed addition example from earlier.

We start with two boxed ints in our context. The expression unboxes them, adds them, and boxes the result.

First, typing rule (Box) trades our box operator for a locked context.

Application rule (App) is standard, we just compute the type of the function and each argument in turn.

Now, we have a locked context with our two boxed variables. Unbox lets us trade this lock here, as long as the expression x is a boxed result.

The variable rule is now allowed to refer to x, since the context is no longer locked up.

There's a bit of mechanical bookkeeping required here, but the rules themselves are relatively simple.

Staging - invalid variable occurrence

$$\frac{\Gamma, \text{Lock} \vdash e : \tau}{\Gamma \vdash \text{box } e : \Box\tau}(\text{Box}) \qquad \frac{\Gamma \vdash e : \Box\tau \qquad \text{Lock} \notin \Gamma'}{\Gamma, \text{Lock}, \Gamma' \vdash \text{unbox } e : \tau}(\text{Unbox})$$

$$\frac{\text{Lock} \notin \Gamma'}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}(\text{Var})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\text{db} : \text{Database}, \text{Lock} \vdash \text{Db.read\_table} : \cdots}(\text{Const}) \qquad \dfrac{\text{Lock} \notin \text{Lock}}{\text{db} : \text{Database}, \text{Lock} \vdash \text{db} : \cdots}(\text{Var})}{\text{db} : \text{Database}, \text{Lock} \vdash \text{Db.read\_table db} : \cdots}(\text{App})}{\text{db} : \text{Database} \vdash \text{box (Db.read\_table db)} : \cdots}(\text{Box})$$

Here's our "bad return" example. We have a variable db of type Database in the context. The expression defers execution until runtime, then calls Db.read_table on the compile-time database.

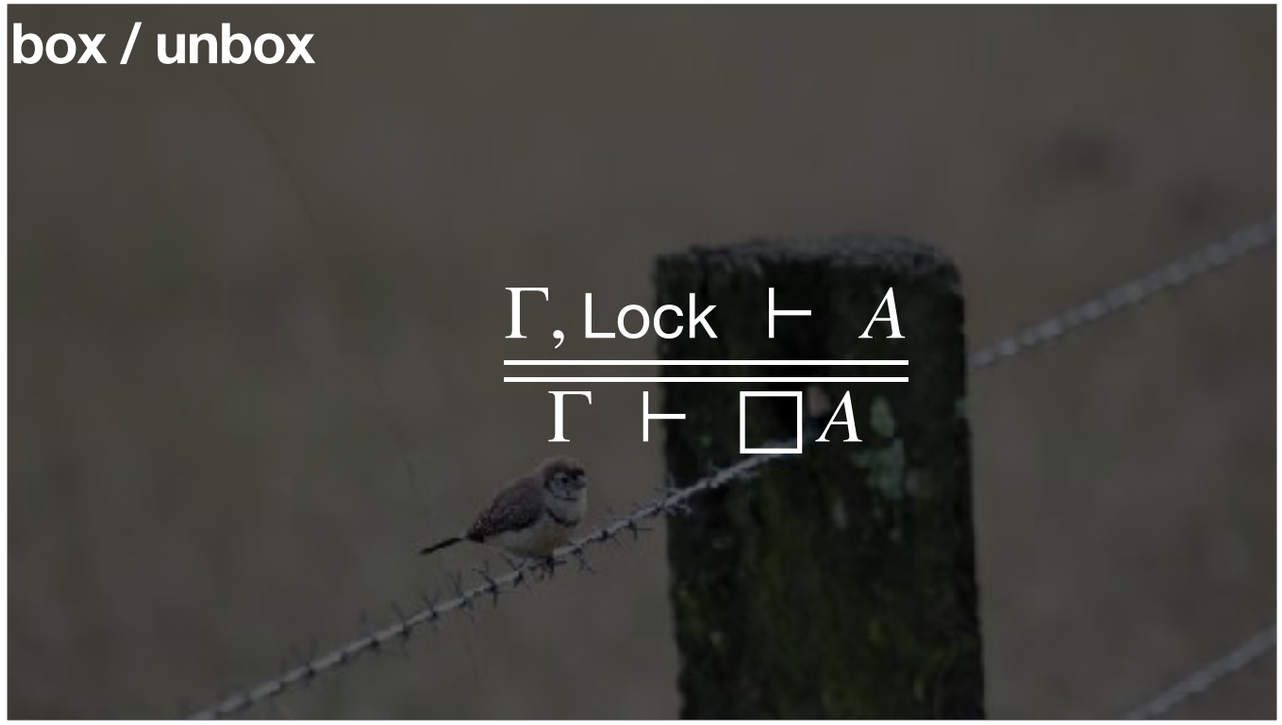
We apply the box rule, which locks up the context.

The app rule requires us to get the type of the function and the argument.

To get the type of the argument, we try to apply the Var rule, but the context is locked up. We can't do it.

On the left, we use the Const rule instead of the Var rule. The Const rule doesn't care that the context is locked up, so it would type check ok if we'd applied it to a different argument.

box / unbox



The key idea that makes this system work is the duality between box and unbox.
Given a boxed value, we can trade it for an unboxed value in a locked context, and vice versa.
This is a nice syntactic property, but it has a semantic meaning as well.

If we interpret box to mean "tomorrow", like going from compiletime to runtime, then we can interpret a locked context to mean "yesterday", going from runtime to compiletime.

With this intuition, we can read the bottom line as "a function from today's context to tomorrow's value".
We can read the top line as "a function from yesterday's context to today's value".
These two views are basically interchangeable, which is why box/unbox work well together.

Icicle in Fitch

k ::= Element | Aggregate

Γ ::= $\cdot \mid \Gamma, x : \tau \mid \Gamma, \mathsf{Lock}_k$

e ::= $\mathsf{box}_k\ e \mid \mathsf{unbox}_k\ e \mid \cdots$

$$\frac{}{\Gamma, x : \tau, \Gamma' \ \vdash\ x : \tau}(\text{Var})$$

$$\frac{\Gamma \ \vdash\ e : \square_k \tau \qquad \mathsf{Lock}_{A,E} \notin \Gamma'}{\Gamma, \mathsf{Lock}_k, \Gamma' \ \vdash\ \mathsf{unbox}_k\ e : \tau}(\text{Unbox})$$

$$\frac{\Gamma, \mathsf{Lock}_k \ \vdash\ e : \tau}{\Gamma \ \vdash\ \mathsf{box}_k\ e : \square_k \tau}(\text{Box})$$

fold : $\tau \to (\square_E \tau \to \square_E \tau) \to \square_A \tau$

We could extend this to Icicle... and support the R axiom by removing the constraint on the Var rule

References

Clouston 2018 - Fitch-style modal lambda calculi

https://arxiv.org/abs/1710.08326v2

Valliappan 2023 - Fitch-style applicative functors (extended abstract)

https://nachivpn.me/r.pdf

https://nachivpn.me/k/

Bahr 2019 - Simply RaTT

https://bahr.io/pubs/entries/bahr19icfp.html