Combining

Dependent, Unique, and Borrow Types

Noé De Santo

Inspired by a true story

Inspired by a true story

(except for the made up parts)

A Tale of Verifying Quicksort

Allow me to take you back when I first learned Rocq, as part of sofware verification class.

A Tale of Verifying Quicksort

Allow me to ta
verification o

Impact of Verification: Software Disasters

- Ariane 5 rocket maiden flight explosion: http://www.inf.ed.ac.uk/teaching/courses/seoc/2008_2009/resources/ariane5.pdf
- Mars Polar orbiter loss: https://en.wikipedia.org/wiki/Mars_Polar_Lander "most likely cause of the mishap was a software error that incorrectly identified vibrations"
- Accidents in various Boeing models (777, 737 MAX, ...)
- Northeast blackout of 2003: https://en.wikipedia.org/wiki/Northeast_blackout_of_2003 (race condition)
- Radio therapy machine Therac-25: https://en.wikipedia.org/wiki/Therac-25

A Tale of Verifying Quicksort

Allow me to take you back when I first learned Rocq, as part of sofware verification class.

Final project was open-ended: we were free to verify **anything**.

A Tale of Verifying Quicksort

Allow me to take you back when I first learned Rocq, as part of sofware verification class.

Final project was open-ended: we were free to verify **anything.**

One of the ideas we considered resulted in an interesting thought:

Quicksort

Verifying Quicksort; or, Playing Pretend

```
Quicksort(A, p, R)
  if p < r
    q = Partition(A, p, R)
    Quicksort(A, p, q-1)
    Quicksort(A, q+1, r)
```

```
Partition(A, p, r)
  x = A[r]
  i = p-1
  for j = p to r-1
    if A[j] ≤ x
      i = i+1
      exchange A[i] with A[j]

  exchange A[i+1] with A[r]
  return i+1
```

Verifying Quicksort; or, Playing Pretend

```
Quicksort(A, p, R)
  if p < r
    q = Partition(A,
    Quicksort(A, p,
    Quicksort(A, q+1, r)
```

```
        i = i+1
        exchange A[i] with A[j]

    exchange A[i+1] with A[r]
    return i+1
```

```
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x]
               ++ qsort (filter (≥ x) xs)
```

[Alexandrescu, Andrei. On Iteration. informIT (2009).]

Verifying Quicksort; or, Playing Pretend

```
Quicksort(A, p, R)
  if p < r
    q = Partition(A, p, R)
    Quicksort(A, p, q-1)
    Quicksort(A, q+1, r)
```

```
Partition(A, p, r)
  x = A[r]
  i = p-1
  for j = p to r-1
    if A[j] ≤ x
      i = i+1
      exchange A[i] with A[j]

  exchange A[i+1] with A[r]
  return i+1
```

Verifying Quicksort; or, Playing Pretend

```
Quicksort(A, p, R)
  if p < r
    A, q = Partition(A, p, R)
    A = Quicksort(A, p, q-1)
    Quicksort(A, q+1, r)
```

```
Partition(A, p, r)
  x = A[r]
  i = p-1
  i, A = for j = p to r-1
    if A[j] ≤ x
      i = i+1
      A = exchange A[i] with A[j]
      yield i, A
  A = exchange A[i+1] with A[r]
  return A, i+1
```

We can pretend that:

1. It is functional by threading A (and i);

Verifying Quicksort; or, Playing Pretend

```
Quicksort(A, p, R)
  if p < r
    A, q = Partition(A, p, R)
    A = Quicksort(A, p, q-1)
    Quicksort(A, q+1, r)
```

```
Partition(A, p, r)
  x = A[r]
  i = p-1
  i, A = for j = p to r-1
    if A[j] ≤ x
      i = i+1
      A = exchange A[i] with A[j]
      yield i, A
  A = exchange A[i+1] with A[r]
  return A, i+1
```

We can pretend that:

1. It is functional by threading A (and i);
2. It is imperative by assuming exchange operates in place.

Verifying Quicksort; or, Playing Pretend

```
Quicksort(A, p, R)
  if p < r
    A, q = Partition(A, p, R)
    A = Quicksort(A, p, q-1)
    Quicksort(A, q+1, r)
```

Functional ⇒ Ease of reasoning
Imperative ⇒ Runtime efficiency

```
Pa
  i = p-1
  i, A = for j = p to r-1
    if A[j] ≤ x
      i = i+1
      A = exchange A[i] with A[j]
      yield i, A
  A = exchange A[i+1] with A[r]
  return A, i+1
```

We can pretend that:

1. It is functional by threading A (and i);
2. It is imperative by assuming exchange operates in place.

Today's plan: Building on this Intuition

Goal: Examine foundations for a proof assistant that allows verification and execution of programs with lightweight mutation.

1. What exactly is going on in our example?
2. How do we build a calculus with all of these ingredients?
3. The metatheory of said calculus.

Today's plan: Building on this Intuition

Goal: Examine foundations for a proof assistant that allows verification and execution of programs with lightweight mutation.

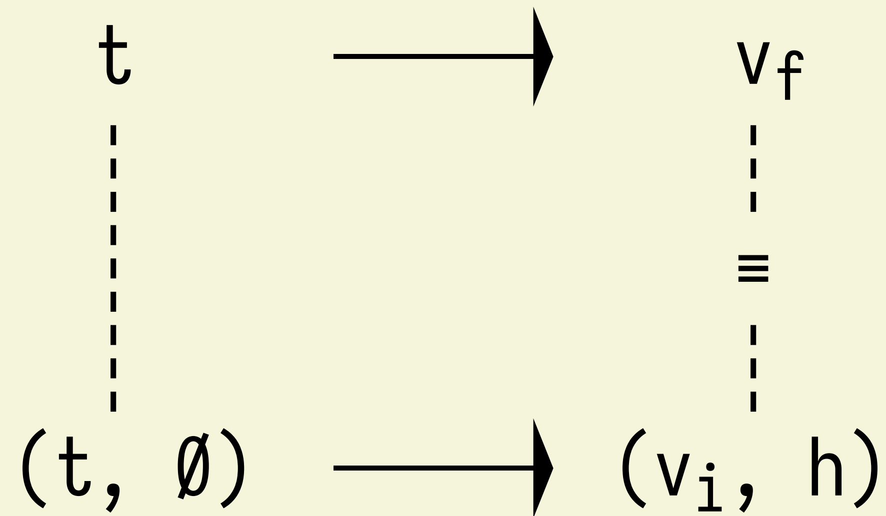
1. What exactly is going on in our example?
2. How do we build a calculus with all of these ingredients?
3. The metatheory of said calculus.

All of this is Work in Progress

Desideratum

To Capture our Intuition: A Pair of Semantics

The intuition was that the program could be read as being both functional and imperative; that implies the existence of operational semantics for both interpretations.



An implicit requirement: Dependent types

Well, that's what proof assistants are (typically) made of...

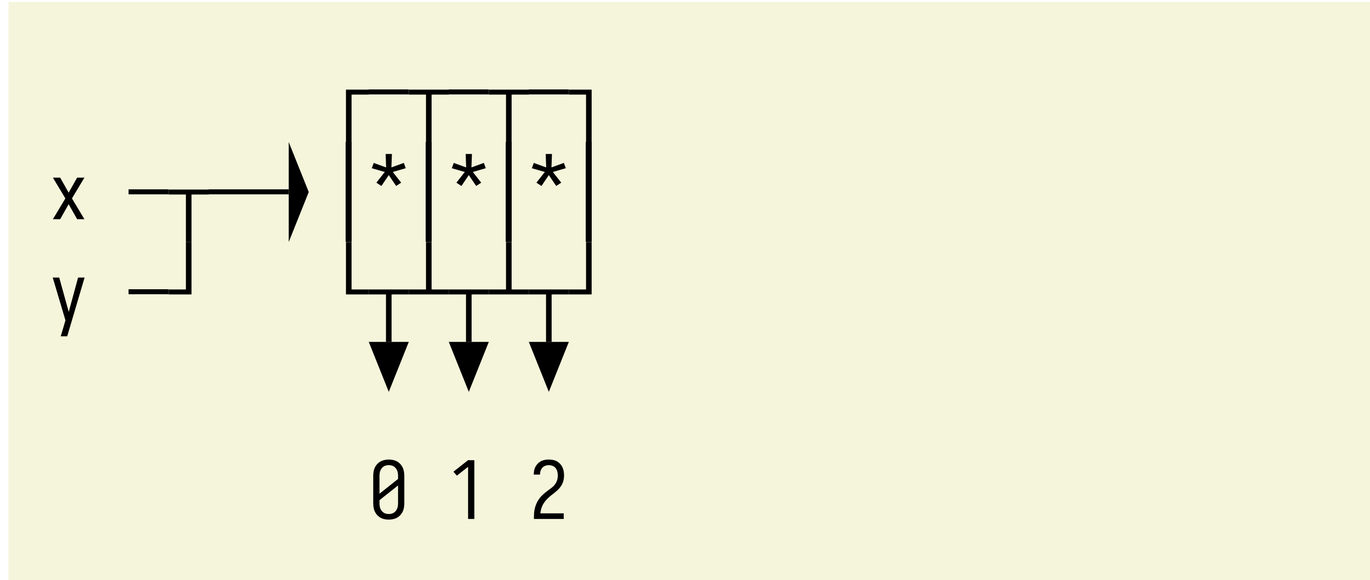
```
let x: Int := 0 in
let eq: x = 0 := eq_refl in
...
```

Functional Mutation: Linear Types

```
x = [0; 1; 2]
y = x
y[0] ← 10
return x, y
```

Functional Mutation: Linear Types

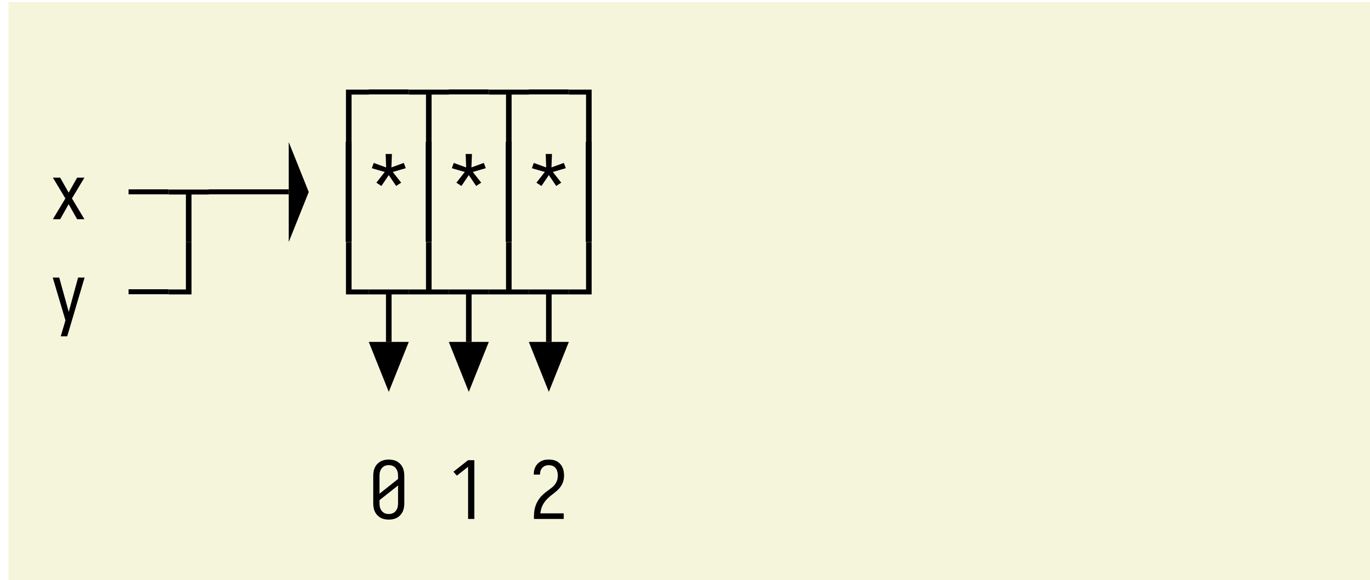
```
x = [0; 1; 2]
y = x
y[0] ← 10
return x, y
```



Imperatively:	[10; 1; 2]	[10; 1; 2]
Functionally:	[0; 1; 2]	[0; 1; 2]

Functional Mutation: Linear Types

```
x = [0; 1; 2]
y = x
y' = y[0] ← 10
return x, y'
```



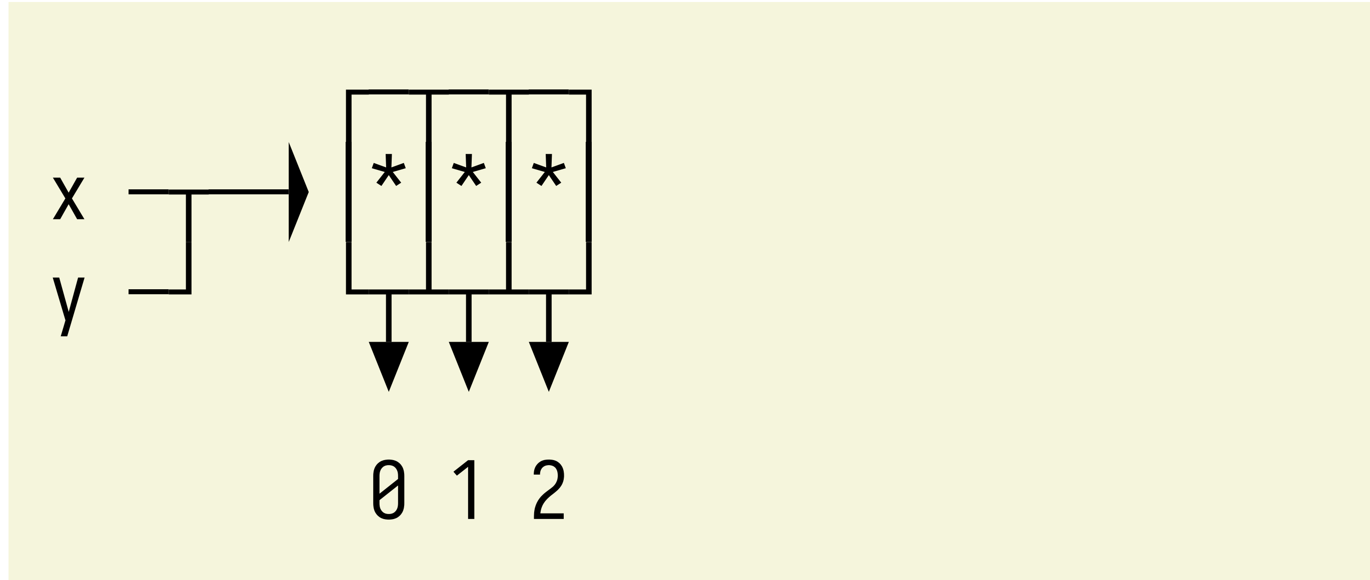
Imperatively:	[10; 1; 2]	[10; 1; 2]
Functionally:	[0; 1; 2]	[10; 1; 2]

Solutions:

- Threading

Functional Mutation: Linear Types

```
x = [0; 1; 2]
y = x
y' = y[0] ← 10
return x, y'
```



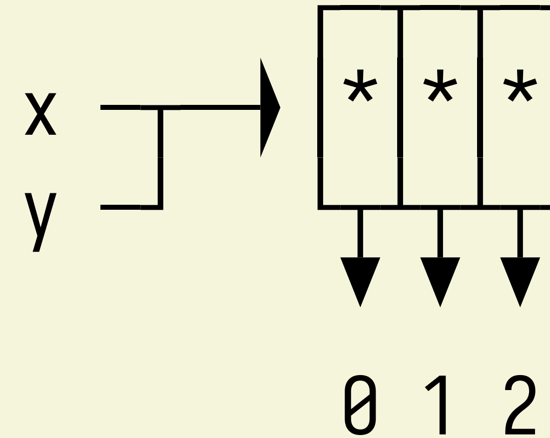
Imperatively:	[10; 1; 2]	[10; 1; 2]
Functionally:	[0; 1; 2]	[10; 1; 2]

Solutions:

- Threading
- Linear Types

Functional Mutation: Linear Types

```
x = [0; 1; 2]
y = x
y' = y[0] ← 10
return y'
```



Imperatively:	~~[10; 1; 2]~~	[10; 1; 2]
Functionally:	~~[0; 1; 2]~~	[10; 1; 2]

Solutions:

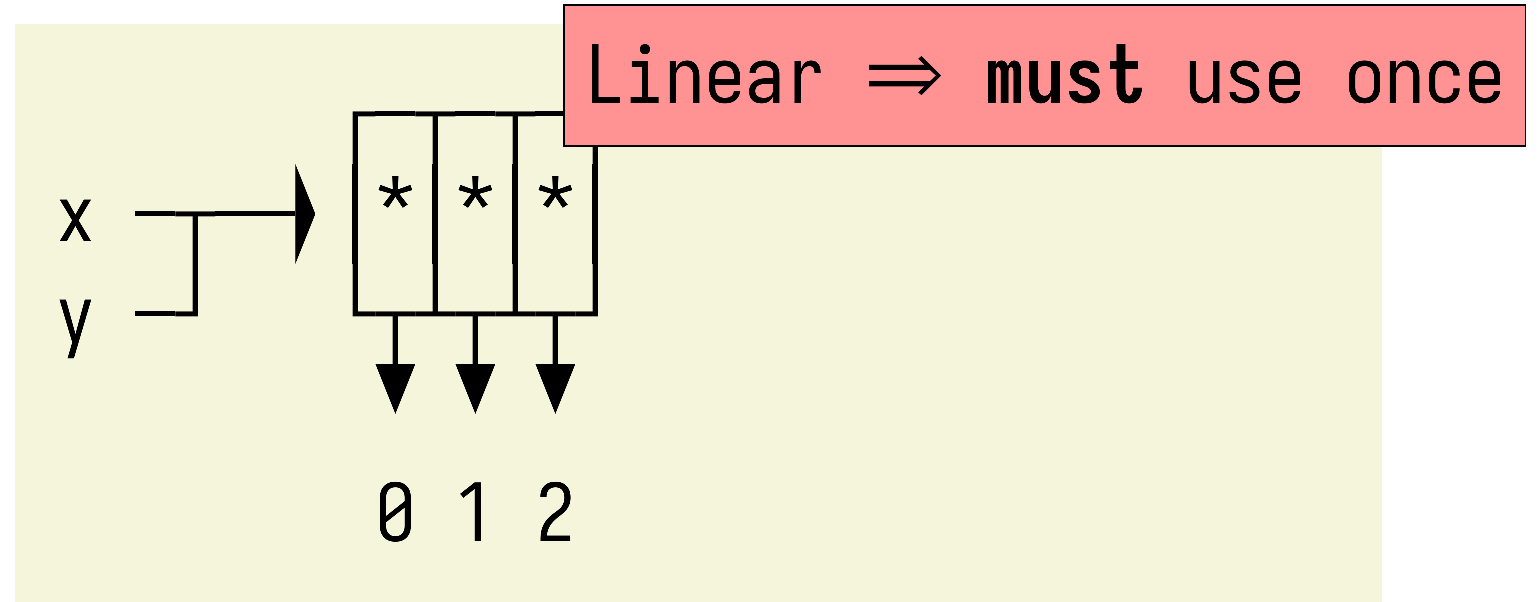
- Threading
- Linear Types

Nothing new: Whole reason linear types were introduced.

P. Wadler. Linear types can change the world!. Programming concepts and methods (1990).

Functional Mutation: Linear Types

```
x = [0; 1; 2]
y = x
y' = y[0] ← 10
return y'
```



Imperatively: ~~[10; 1; 2]~~ [10; 1; 2]
Functionally: ~~[0; 1; 2]~~ [10; 1; 2]

Solutions:

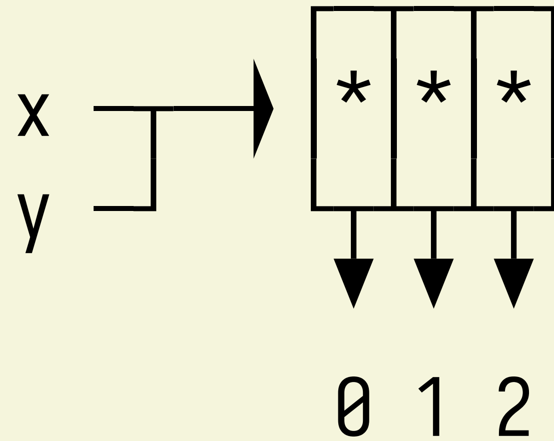
- Threading
- Linear Types

Nothing new: Whole reason linear types were introduced.

P. Wadler. Linear types can change the world!. Programming concepts and methods (1990).

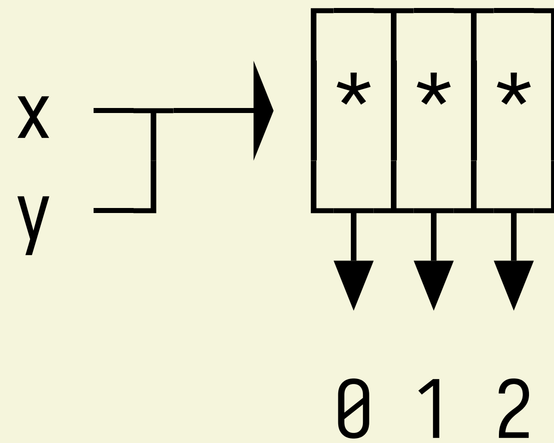
Functional Mutation: ~~Linear~~ Unique Types

As the previous slide kind of hinted at, linearity is just a discipline to attain our actual goal: uniqueness.



Functional Mutation: ~~Linear~~ Unique Types

As the previous slide kind of hinted at, linearity is just a discipline to attain our actual goal: uniqueness.

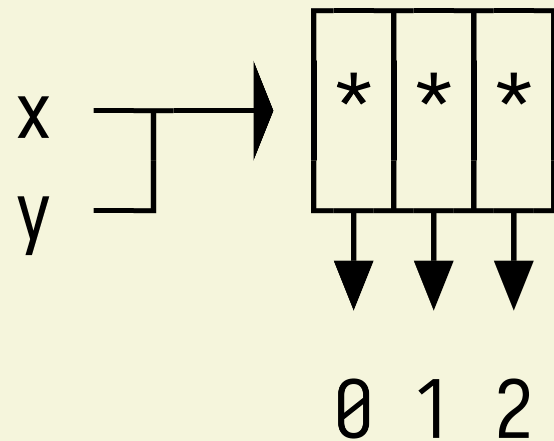


Uniqueness is a better way to reason about whether a language feature makes sense.

Functional Mutation: ~~Linear~~ Unique Types

As the previous slide kind of hinted at,
attain our actual goal: uniqueness.

A distinction can be made between linear & unique types;
our "linear" types will be both.



Uniqueness is a better way to reason about whether a language feature makes sense.

[D. Marshall, M. Vollmer, D. Orchard. Linearity and uniqueness: An entente cordiale. ESOP (2022).]

Linearity is hard to work with: Unrestriced types

```
f = λ x, 2*x
f (f 10)
```

This programs does not follow the linear discipline...

Linearity is hard to work with: Unrestriced types

```
f = λ x, 2*x
f (f 10)
```

This programs does not follow the linear discipline...

... and we should allow it.

Linearity is hard to work with: Borrow types

```
Partition(A, p, r)
  x = A[r]
  i = p-1
  i, A = for j = p to r-1
    if A[j] ≤ x
      i = i+1
      A = exchange A[i] with A[j]
      yield i, A
  A = exchange A[i+1] with A[r]
  return A, i+1
```

Linearity is hard to work with: Borrow types

```
Partition(A, p, r)
  x = A[r]
  i = p-1
  i, A = for j = p to r-1
    if A[j] ≤ x
      i = i+1
      A = exchange A[i] with A[j]
      yield i, A
  A = exchange A[i+1] with A[r]
  return A, i+1
```

The first A is used three times!

Linearity is hard to work with: Borrow types

Borrows in a nutshell:

```
x = 0                                           // Some linear variable
borrow x {
  y = (x, x)                                    // Usable many times
  if x == 10                                    // Inspectable
  then ... else ...
  x' = x ← 10                                   // But not mutable
}
x' = x ← 10                                     // Mutable again after the borrow
```

To Write a Spec: Runtime Irrelevance

```
forall A A' i j, A' = exchange A[i] A[j] → A[i] = A'[j]
```

This also obviously breaks linearity again, but this would nonetheless be useful.

Desideratum: Summary

- Pair of semantics:
  - Functional for reasoning
  - Imperative for actual execution
- Proof Assistant $\Rightarrow$ Dependent Types
- Mutation $\Rightarrow$ Linearity/uniqueness
- Linearity/uniqueness is too resrictive $\Rightarrow$ Escape hatches
  - Unrestricted types
  - Borrow types
- Specification $\Rightarrow$ Runtime irrelevance

No element here is novel, but the combination is.

From our Desideratum to

A Core Calculus

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                        Terms
    | ⟨⟩       | let ⟨⟩ := t1 in t2             Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2         Pair
    | λx, t    | t1 t2                         Function
    | letr x := t1 in t2                       Borrow
```

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                                   Terms
    | ⟨⟩       | let ⟨⟩ := t1 in t2                         Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2                     Pair
    | λx, t    | t1 t2                                   Function
    | letr x := t1 in t2                                   Borrow
```

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                            Terms
    | ⟨⟩       | let ⟨⟩ := t1 in t2                Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2            Pair
    | λx, t    | t1 t2                             Function
    | letr x := t1 in t2                           Borrow
```

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                       Terms
    | ⟨⟩       | let ⟨⟩ := t1 in t2            Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2        Pair
    | λx, t    | t1 t2                         Function
    | letr x := t1 in t2                       Borrow
```

```
T ::= Unit[μ] | T1 *[μ]* T2 | T1 =[μ]⇒ T2 | &T        Types
```

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                         Terms
    | ⟨⟩         | let ⟨⟩ := t1 in t2             Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2         Pair
    | λx, t      | t1 t2                        Function
    | letr x := t1 in t2                        Borrow
```

```
T ::= Unit[μ] | T1 *[μ]* T2 | T1 =[μ]⇒ T2 | &T        Types
```

```
μ ::= 1 | ω | 0                                 Usage/Mode
```

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                          Terms
    | ⟨⟩       | let ⟨⟩ := t1 in t2              Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2          Pair
    | λx, t    | t1 t2                           Function
    | letr x := t1 in t2                         Borrow
```

```
T ::= Unit[μ] | T1 *[μ]* T2 | T1 =[μ]⇒ T2 | &T    Types
```

```
μ ::= 1 | ω | 0                                   Usage/Mode
```

Syntax

The stated goal was a proof assistant, but that's a bit too ambitious.
Let's start with the STLC.

```
t ::= x                                          Terms
    | ⟨⟩       | let ⟨⟩ := t1 in t2              Unit
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2          Pair
    | λx, t    | t1 t2                           Function
    | letr x := t1 in t2                         Borrow
```

```
T ::= Unit[μ] | T1 *[μ]* T2 | T1 =[μ]⇒ T2 | &T   Types
```

```
μ ::= 1 | ω | 0                                  Usage/Mode
```

Syntax

The stated goal was a proof assistant, but that'
Let's start with the STLC.

```
t ::= x                                                          s
    | ⟨⟩        | let ⟨⟩ := t1 in t2                              t
    | ⟨t1, t2⟩ | let ⟨x, y⟩ := t1 in t2                        Pair
    | λx, t     | t1 t2                                     Function
    | letr x := t1 in t2                                      Borrow
```

```
T ::= Unit[μ] | T1 *[μ]* T2 | T1 =[μ]⇒ T2 | &T                Types
```

```
μ ::= 1 | ω | 0                                          Usage/Mode
```

```
Usage(Unit[μ])       = μ
Usage(A *[μ]* B)     = μ
Usage(A =[μ]⇒ B)     = μ
Usage(&A)            = ω
```

Typing: Key Ideas

$$\frac{\begin{array}{c}\Gamma \vdash t_1 \in T_1 \\ \Gamma \vdash t_2 \in T_2\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

Typing: Key Ideas

$$\frac{\Gamma \vdash t_1 \in T_1 \qquad \Gamma \vdash t_2 \in T_2}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,n⟩ in
...
```

I'll use some extensions (let-bindings & integers) for clarity.

Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,n⟩ in
...
```

I'll use some extensions (let-bindings & integers) for clarity.

Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

$\texttt{Usage}(T)$	
0	$T \rightsquigarrow T + T$
ω	$T \rightsquigarrow T + T$
1	$T \rightsquigarrow T + \texttt{observe}(T)$
1	$T \rightsquigarrow \texttt{observe}(T) + T$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,n⟩ in
...
```

Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

Usage (T)	
0	$T \rightsquigarrow T + T$
ω	$T \rightsquigarrow T + T$
1	$T \rightsquigarrow T + \texttt{observe}\,(T)$
1	$T \rightsquigarrow \texttt{observe}\,(T) + T$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,n⟩ in
...
```

```
observe(Nat[1]) = Nat[0]
```

Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

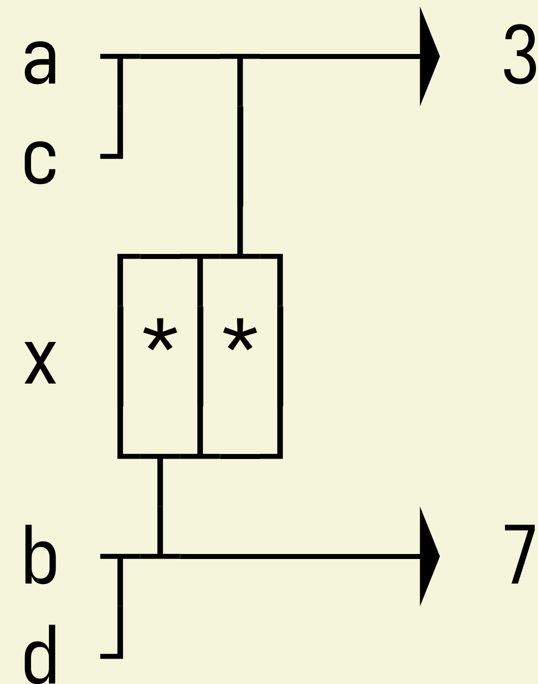
Usage (T)	
0	$T \rightsquigarrow T + T$
ω	$T \rightsquigarrow T + T$
1	$T \rightsquigarrow T + \texttt{observe}\,(T)$
1	$T \rightsquigarrow \texttt{observe}\,(T) + T$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,7⟩ in
let ⟨a,b⟩ := x in
let ⟨c,d⟩ := x in
...
```

Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,7⟩ in
let ⟨a,b⟩ := x in
let ⟨c,d⟩ := x in
...
```



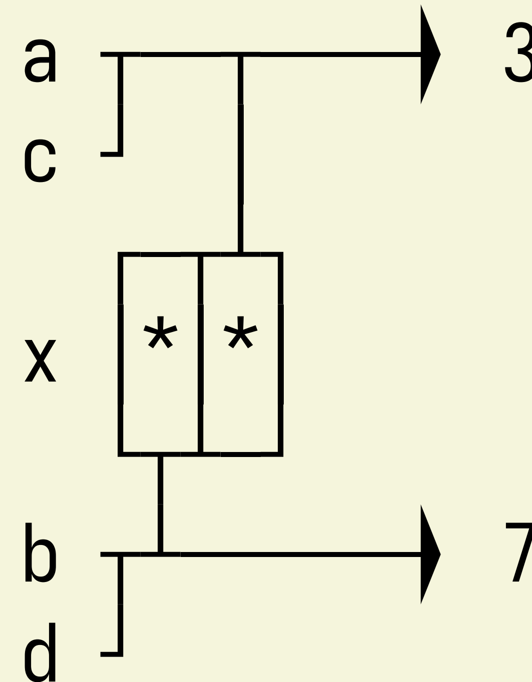
Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{l}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array} \qquad \begin{array}{l}\texttt{Usage}\,(T_1) \sqsubseteq \mu \\ \texttt{Usage}\,(T_2) \sqsubseteq \mu\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

Containment Order:

$$0 \sqsubseteq \omega \sqsubseteq 1$$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,7⟩ in
let ⟨a,b⟩ := x in
let ⟨c,d⟩ := x in
...
```



Typing: Key Ideas

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in T_1 \\ \Gamma_2 \vdash t_2 \in T_2\end{array} \qquad \begin{array}{c}\texttt{Usage}\,(T_1) \sqsubseteq \mu \\ \texttt{Usage}\,(T_2) \sqsubseteq \mu\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \in T_1 *[\mu]* T_2}$$

Containment Order:

$$0 \sqsubseteq \omega \sqsubseteq 1$$

```
let n: Nat[1] := 3 in
let x: Nat[1] *[ω]* Nat[1] := ⟨n,7⟩ i
let ⟨a,b⟩ := x in
let ⟨c,d⟩ := x in
...
```

These constrains also generate a well-formedness judgment for types:

$$\frac{\begin{array}{c}\vdash T_1 \in * \\ \vdash T_2 \in *\end{array} \qquad \begin{array}{c}\texttt{Usage}\,(T_1) \sqsubseteq \mu \\ \texttt{Usage}\,(T_2) \sqsubseteq \mu\end{array}}{\vdash T_1 *[\mu]* T_2 \in *}$$

Typing: Borrows

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

$$\frac{T \rightsquigarrow U + V}{T \overset{\&}{\rightsquigarrow} U + V}$$

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

$$\frac{T \rightsquigarrow U + V}{T \overset{\&}{\rightsquigarrow} U + V}$$

$$\frac{\texttt{Usage}(T) \neq 0}{T \overset{\&}{\rightsquigarrow} \texttt{\&}T + T}$$

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]

in
(z, x)
```

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]
  5
in
(z, x)
```

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]
  match z with | 0 ⇒ 3 | S _ ⇒ 5
in
(z, x)
```

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]
  (z, z)
in
(z, x)
```

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array} \qquad \texttt{borrowing?}\,(S) = \texttt{false}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]
  (z, z)                         // Error: S (= &Nat[1] *[ω]* &Nat[1]) is borrowing
in
(z, x)
```

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array} \qquad \texttt{borrowing?}(S) = \texttt{false}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]
  match z with | 0 ⇒ 3 | S i ⇒ i
in
(z, x)
```

Typing: Borrows

$$\frac{\Gamma \overset{\&}{\rightsquigarrow} \Gamma_1 + \Gamma_2 \qquad \begin{array}{rl} \Gamma_1 & \vdash t_1 \in S \\ x : S, \Gamma_2 & \vdash t_2 \in T \end{array} \qquad \texttt{borrowing?}(S) = \texttt{false}}{\Gamma \vdash \texttt{letr x := } t_1 \texttt{ in } t_2 \in T}$$

```
let z: Nat[1] := 10 in
letr x := // Borrow z: &Nat[1]
  match z with | 0 ⇒ 3 | S i ⇒ i          // Error: i has type &Nat[1]
in
(z, x)
```

Operational Semantics

Functional ($\longmapsto_f$): Usual (substitution-based) CBV.

Imperative ($\longmapsto_i$): Heap-based.

Operational Semantics

Functional ($\longmapsto_f$): Usual (substitution-based) CBV.

$$\frac{\text{Value } v_1 \qquad \text{Value } v_2}{\text{Value } \langle v_1, v_2 \rangle}$$

$$\texttt{let } \langle \texttt{x,y} \rangle \texttt{ := } \langle v_1, v_2 \rangle \texttt{ in } t \longmapsto_f [x \mapsto v_1, y \mapsto v_2]t$$

Imperative ($\longmapsto_i$): Heap-based.

Operational Semantics

Functional ($\longmapsto_f$): Usual (substitution-based) CBV.

$$\frac{\text{Value } v_1 \qquad \text{Value } v_2}{\text{Value } \langle v_1, v_2 \rangle}$$

$$\texttt{let } \langle \texttt{x,y} \rangle \texttt{ := } \langle v_1, v_2 \rangle \texttt{ in } t \quad \longmapsto_f \quad [x \mapsto v_1, y \mapsto v_2]t$$

Imperative ($\longmapsto_i$): Heap-based.

$$\frac{}{\text{Value } \ell}$$

Operational Semantics

Functional ($\longmapsto_f$): Usual (substitution-based) CBV.

$$\frac{\text{Value } v_1 \qquad \text{Value } v_2}{\text{Value } \langle v_1, v_2 \rangle}$$

$$\texttt{let } \langle \texttt{x,y} \rangle \texttt{ := } \langle v_1, v_2 \rangle \texttt{ in } t \;\longmapsto_f\; [x \mapsto v_1, y \mapsto v_2]t$$

Imperative ($\longmapsto_i$): Heap-based.

$$\frac{}{\text{Value } \ell}$$

$$\langle \ell_1, \ell_2 \rangle \circ \mathfrak{h} \;\overset{\ell \text{ fresh}}{\longmapsto_i}\; \ell \circ \mathfrak{h}, \ell \mapsto \langle \ell_1, \ell_2 \rangle$$

Operational Semantics

Functional ($\longmapsto_f$): Usual (substitution-based) CBV.

$$\frac{\text{Value } v_1 \qquad \text{Value } v_2}{\text{Value } \langle v_1, v_2 \rangle}$$

$$\texttt{let } \langle \texttt{x,y} \rangle \texttt{ := } \langle v_1, v_2 \rangle \texttt{ in } t \longmapsto_f [x \mapsto v_1, y \mapsto v_2]t$$

Imperative ($\longmapsto_i$): Heap-based.

$$\frac{}{\text{Value } \ell}$$

$$\langle \ell_1, \ell_2 \rangle \circ \mathfrak{h} \xrightarrow{\ell \text{ fresh}}_i \ell \circ \mathfrak{h}, \ell \mapsto \langle \ell_1, \ell_2 \rangle$$

$$\texttt{let } \langle \texttt{x, y} \rangle \texttt{ := } \ell \texttt{ in } t \circ \mathfrak{h} \xrightarrow{\mathfrak{h}\ell = \langle \ell_1, \ell_2 \rangle}_i [x \mapsto \ell_1, y \mapsto \ell_2]t \circ \mathfrak{h}$$

Wait, Where is the Mutation?

We could add array, but for this presentation, we'll go with something simpler:

```
t ::= ...                                    Terms
    | n | t ← n | ...                      Natural
```

```
T ::= ... | Nat[μ]                           Types
```

Wait, Where is the Mutation?

We could add array, but for this presentation, we'll go with something simpler:

```
t ::= ...                                              Terms
    | n | t ← n | ...                                Natural
```

```
T ::= ... | Nat[μ]                                     Types
```

$$\frac{\Gamma \vdash t \in \mathbf{Nat[1]}}{\Gamma \vdash t \texttt{ <- } n \in \mathbf{Nat[1]}}$$

Wait, Where is the Mutation?

We could add array, but for this presentation, we'll go with something simpler:

```
t ::= ...                                        Terms
    | n | t ← n | ...                            Natural
```

```
T ::= ... | Nat[μ]                               Types
```

$$\frac{\Gamma \vdash t \in \texttt{Nat[1]}}{\Gamma \vdash t \texttt{ <- } n \in \texttt{Nat[1]}}$$

$$v \texttt{ <- } n \longmapsto_f n$$

Wait, Where is the Mutation?

We could add array, but for this presentation, we'll go with something simpler:

```
t ::= ...                                   Terms
    | n | t ← n | ...                       Natural
```

```
T ::= ... | Nat[μ]                          Types
```

$$\frac{\Gamma \vdash t \in \mathbf{Nat[1]}}{\Gamma \vdash t \texttt{ <- } n \in \mathbf{Nat[1]}}$$

$$v \texttt{ <- } n \longmapsto_f n$$

$$\ell \leftarrow \mathrm{n} \circ \mathfrak{h}, \ell \mapsto m \longmapsto_i \ell \circ \mathfrak{h}, \ell \mapsto n$$

Wait, Where is the Mutation?

We could add array, but for this presentation, we'll go with something simpler:

```
t ::= ...                                  Terms
    | n | t ← n | ...                     Natural
```

```
T ::= ... | Nat[μ]                         Types
```

$$\frac{\Gamma \vdash t \in \texttt{Nat[1]}}{\Gamma \vdash t \texttt{ <- } n \in \texttt{Nat[1]}}$$

$$\frac{\Gamma \vdash t \in \texttt{Nat[0]}}{\Gamma \vdash t \texttt{ <- } n \in \texttt{Nat[0]}}$$

$$v \texttt{ <- } n \longmapsto_f n$$

$$\ell \leftarrow \texttt{n} \circ \mathfrak{h}, \ell \mapsto m \longmapsto_i \ell \circ \mathfrak{h}, \ell \mapsto n$$

Wait, Where is the Mutation? (Redux)

```
t ::= ...                                              Terms
    | [t1; ..; tn] | swap i, j in t | ...              Array
```

```
T ::= ... | Array[μ; T]                                Types
```

Wait, Where is the Mutation? (Redux)

```
t ::= ...                                              Terms
    | [t1; ..; tn] | swap i, j in t | ...              Array
```

```
T ::= ... | Array[μ; T]                                Types
```

$$\frac{\begin{array}{c}\Gamma \leadsto \Gamma_1 + \Delta \\ \Delta \leadsto \Gamma_2 + \Gamma_3\end{array} \qquad \begin{array}{c}\Gamma_1 \vdash t_1 \in \texttt{Nat}[\omega] \\ \Gamma_2 \vdash t_2 \in \texttt{Nat}[\omega]\end{array} \qquad \Gamma_3 \vdash t_3 \in \texttt{Array}[\mu;\ \texttt{T}]}{\Gamma \vdash \texttt{swap}\ t_1, t_2\ \texttt{in}\ t_3}$$

Wait, Where is the Mutation? (Redux)

```
t ::= ...                                          Terms
    | [t1; ..; tn] | swap i, j in t | ...          Array
```

```
T ::= ... | Array[μ; T]                            Types
```

$$\texttt{swap}\ \ell_i,\ \ell_j\ \texttt{in}\ \ell \ \circ\ \mathfrak{h}, \ell_i \mapsto i, \ell_j \mapsto j, \ell \mapsto [..;\ \underset{i^{\text{th}}}{\ell_i^a}\ ;..;\ \underset{j^{\text{th}}}{\ell_j^a}\ ;..]$$

$$\longmapsto_i$$

$$\ell \ \circ\ \mathfrak{h}, \ell_i \mapsto i, \ell_j \mapsto j, \ell \mapsto [..;\ \underset{i^{\text{th}}}{\ell_j^a}\ ;..;\ \underset{j^{\text{th}}}{\ell_i^a}\ ;..]$$

Borrows require Call-by-Value

```
let z: Nat[1] := 0 in
letr x := // Borrow z: &Nat[1]
    match z with | 0 ⇒ 10 | S _ ⇒ 20
in
let z' := z ← 1 in
z' + x
```

Borrows require Call-by-Value

```
let z: Nat[1] := 0 in
letr x := // Borrow z: &Nat[1]
    match z with | 0 ⇒ 10 | S _ ⇒ 20
in
let z' := z ← 1 in
z' + x
```

	Functional	Imperative
CBV	11	11
CBN	11	21

Metatheory

Metatheory: The Plan

- Functional (Syntactic) Safety: Progress + Preservation
- Semantic Equivalence: Functional & Imperative are "the same"

Weakening lemmas

It is well-known that linear systems drop the weakening property:

Weakening (does not hold!!!)

If $\Gamma_1, \Gamma_2 \vdash t \in T$, then $\Gamma_1, x : A, \Gamma_2 \vdash t \in T$.

Weakening lemmas

It is well-known that linear systems drop the weakening property:

Weakening (does not hold!!!)

If $\Gamma_1, \Gamma_2 \vdash t \in T$, then $\Gamma_1, x : A, \Gamma_2 \vdash t \in T$.

But this systems admits other forms of weakening:

0-Weakening

If $\Gamma_1, \Gamma_2 \vdash t \in T$, and $\mathbf{Usage}\,(A) = 0$, then $\Gamma_1, x : A, \Gamma_2 \vdash t \in T$

Weakening lemmas

It is well-known that linear systems drop the weakening property:

Weakening (does not hold!!!)

If $\Gamma_1, \Gamma_2 \vdash t \in T$, then $\Gamma_1, x : A, \Gamma_2 \vdash t \in T$.

But this systems admits other forms of weakening:

0-Weakening

If $\Gamma_1, \Gamma_2 \vdash t \in T$, and $\textbf{Usage}\left(A\right) = 0$, then $\Gamma_1, x : A, \Gamma_2 \vdash t \in T$

ω-Weakening

If $\Gamma_1, x : \textbf{observe}\left(A\right), \Gamma_2 \vdash t \in T$, and $\textbf{Usage}\left(A\right) = \omega$,
then $\Gamma_1, x : A, \Gamma_2 \vdash t \in T$

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \rightsquigarrow A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \leadsto A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \leadsto \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Typing Split

$$y : \mathtt{Nat[1]}, z : \mathtt{Nat[1]} \quad \vdash \langle y, z \rangle \in \quad \mathtt{Nat[1]\ *[1]*\ Nat[1]}$$

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \leadsto A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \leadsto \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Typing Split

$$y : \texttt{Nat[1]}, z : \texttt{Nat[1]} \quad \vdash \langle y, z \rangle \in \quad \texttt{Nat[1] *[1]* Nat[1]}$$

$$\leadsto$$

$$\texttt{Nat[0] *[0]* Nat[0]}$$

$$+$$

$$\texttt{Nat[1] *[1]* Nat[1]}$$

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \rightsquigarrow A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Typing Split

$$
\begin{array}{ccccc}
y : \texttt{Nat[1]}, z : \texttt{Nat[1]} & \vdash \langle y, z \rangle \in & \texttt{Nat[1]} & \texttt{*[1]*} & \texttt{Nat[1]} \\
\rightsquigarrow & & & \rightsquigarrow & \\
y : \texttt{Nat[0]}, z : \texttt{Nat[0]} & & \texttt{Nat[0]} & \texttt{*[0]*} & \texttt{Nat[0]} \\
+ & & & + & \\
y : \texttt{Nat[1]}, z : \texttt{Nat[1]} & & \texttt{Nat[1]} & \texttt{*[1]*} & \texttt{Nat[1]}
\end{array}
$$

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \rightsquigarrow A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Typing Split

$$
\begin{array}{lcl}
y : \texttt{Nat[1]}, z : \texttt{Nat[1]} & \vdash \langle y, z\rangle \in & \texttt{Nat[1] *[1]* Nat[1]} \\
\rightsquigarrow & & \rightsquigarrow \\
y : \texttt{Nat[0]}, z : \texttt{Nat[0]} & \vdash \langle y, z\rangle \in & \texttt{Nat[0] *[0]* Nat[0]} \\
+ & & + \\
y : \texttt{Nat[1]}, z : \texttt{Nat[1]} & \vdash \langle y, z\rangle \in & \texttt{Nat[1] *[1]* Nat[1]}
\end{array}
$$

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \rightsquigarrow A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Typing Split

Substitution split

```
let x := t in ⟨l, r⟩
```

Splitting Substitutions

Typing Split

If $\Gamma \vdash t \in A$ and $A \rightsquigarrow A_1 + A_2$,
then there exist Γ_1, Γ_2 such that $\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2$,
$\Gamma_1 \vdash t \in A_1$, and $\Gamma_2 \vdash t \in A_2$.

Typing Split

Substitution split

```
⟨[x↦t]l, [x↦t]r⟩
```

Substitution

Preservation under Substitution

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T$, $\Delta \vdash s \in S$,

then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

Substitution

Preservation under Substitution

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T,\ \Delta \vdash s \in S$,
and $\forall x, \texttt{Usage}\,(\Delta x) \sqsubseteq \texttt{Usage}\,(S)$,
then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

Substitution

Preservation under Substitution

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T$, $\Delta \vdash s \in S$,
and $\forall x, \textbf{Usage}\left(\Delta x\right) \sqsubseteq \textbf{Usage}\left(S\right)$,
then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

That extra assumption is a bit scary...

Substitution

Preservation under Substitution

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T,\ \Delta \vdash s \in S$,
and $\forall x, \mathbf{Usage}\,(\Delta x) \sqsubseteq \mathbf{Usage}\,(S)$,
then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

That extra assumption is a bit scary... but is actually fine.

Substitution

Preservation under Substitution

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T$, $\Delta \vdash s \in S$,
and $\forall x, \textbf{Usage}\,(\Delta x) \sqsubseteq \textbf{Usage}\,(S)$,
then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

That extra assumption is a bit scary... but is actually fine.

Value repacking

If $\Gamma \vdash v \in T$, then there exists Δ such that
$\forall x, \textbf{Usage}\,(\Delta x) \sqsubseteq \textbf{Usage}\,(T)$ and $\Delta \vdash v \in T$.

Substitution

Preservation under Substitution

Another reason we are limited to CBV.

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T$, $\Delta \vdash s \in S$,
and $\forall x, \textbf{Usage}\,(\Delta x) \sqsubseteq \textbf{Usage}\,(S)$,
then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

That extra assumption is a bit scary... but is actually fine.

Value repacking

If $\Gamma \vdash v \in T$, then there exists Δ such that
$\forall x, \textbf{Usage}\,(\Delta x) \sqsubseteq \textbf{Usage}\,(T)$ and $\Delta \vdash v \in T$.

Substitution

Preservation under Substitution

Another reason we are limited to CBV.

If $\Gamma_1, x : S, \Gamma_2 \vdash t \in T$, $\Delta \vdash s \in S$,
and $\forall x, \mathbf{Usage}\,(\Delta x) \sqsubseteq \mathbf{Usage}\,(S)$,
then $\Gamma_1, \Delta, \Gamma_2 \vdash [x \mapsto s]t \in T$

That extra assumption is a bit scary... but is actually fine.

Value repacking

If $\Gamma \vdash v \in T$, then there exists Δ such that
$\forall x, \mathbf{Usage}\,(\Delta x) \sqsubseteq \mathbf{Usage}\,(T)$ and $\Delta \vdash v \in T$.
Additionally, $\forall x$, $(\Delta x = \Gamma x)$ or $(\Delta x = \mathbf{observe}\,(\Gamma x))$.

Borrow Elimination

Borrow Elimination

If $\Gamma \vdash v \in T$ and $\texttt{borrowing?}\,(T) = \texttt{false}$, then $\texttt{WithoutBorrows}\,(\Gamma) \vdash v \in T$

Borrow Elimination

Borrow Elimination

If $\Gamma \vdash v \in T$ and $\textbf{borrowing?}\,(T) = \textbf{false}$, then $\textbf{WithoutBorrows}\,(\Gamma) \vdash v \in T$

Eliminating borrows

```
let x: Int[1] := 0 in
letr y := // x: &Int[1]
  let z := match x with ...
  ...
in
⟨x, y⟩
```

Borrow Elimination

Borrow Elimination

If $\Gamma \vdash v \in T$ and **borrowing?** $(T) =$ **false**, then **WithoutBorrows** $(\Gamma) \vdash v \in T$

Eliminating borrows

```
let x: Int[1] := 0 in
letr y := // x: &Int[1]


  v
in
⟨x, y⟩
```

Borrow Elimination

Borrow Elimination

If $\Gamma \vdash v \in T$ and $\texttt{borrowing?}(T) = \texttt{false}$, then $\texttt{WithoutBorrows}(\Gamma) \vdash v \in T$

Eliminating borrows

```
let x: Int[1] := 0 in
letr y := // x: Int[0]


  v
in
⟨x, y⟩
```

(Functional) Safety

Progress

If $\vdash t \in T$, then either $\mathrm{Value}\, t$ or $\exists t', t \longmapsto_f t'$

Preservation

If $\Gamma \vdash t \in T$, and $t \longmapsto_f t'$, then $\Gamma \vdash t' \in T$.

(Functional) Safety

Progress

If $\vdash t \in T$, then either $\mathrm{Value}\, t$ or $\exists t', t \longmapsto_f t'$

Preservation

If $\Gamma \vdash t \in T$, and $t \longmapsto_f t'$, then $\Gamma \vdash t' \in T$.

Safety

If $\Gamma \vdash t \in T$, and $t \longmapsto_f^* t'$, then either $\mathrm{Value}\, t'$ or $\exists t'', t' \longmapsto_f t''$

Relating the Semantics

1. Define a logical relation

$$\mathbb{V} : \mathbf{Type} \to \mathbf{HeapTyping} \to \mathbf{Val} \to \mathbf{Loc} \to \mathbf{Heap} \to \mathbb{P}$$

for values.

Relating the Semantics

1. Define a logical relation

$$\mathbb{V} : \mathbf{Type} \to \mathbf{HeapTyping} \to \mathbf{Val} \to \mathbf{Loc} \to \mathbf{Heap} \to \mathbb{P}$$

   for values.
   The HeapTyping achieves a role very similar to typing contexts.

Relating the Semantics

1. Define a logical relation

   $$\mathbb{V} : \mathbf{Type} \to \mathbf{HeapTyping} \to \mathbf{Val} \to \mathbf{Loc} \to \mathbf{Heap} \to \mathbb{P}$$

   for values.
   The HeapTyping achieves a role very similar to typing contexts.
2. Lift the relation to whole computations

Relating the Semantics

1. Define a logical relation

$$\mathbb{V} : \textbf{Type} \to \textbf{HeapTyping} \to \textbf{Val} \to \textbf{Loc} \to \textbf{Heap} \to \mathbb{P}$$

   for values.
   The HeapTyping achieves a role very similar to typing contexts.

2. Lift the relation to whole computations
3. Show that, if one removes all 0-usage computations from a well-typed t, producing t', then these two computations are related by the relation.

Extending this Calculus

Making It Dependent

The next target would be LF; this requires two ingredients:

0-usage:

Type equivalence:

Making It Dependent

The next target would be LF; this requires two ingredients:

0-usage:

```
let x: Nat[1] := 0 in
let p: x = 0 := eq_refl in
let x' := x ← 1 in
...
```

Type equivalence:

Making It Dependent

The next target would be LF; this requires two ingredients:

0-usage:

```
let x: Nat[1] := 0 in
let p: x = 0 := eq_refl in                    // x: Nat[0]
let x' := x ← 1 in                            // x: Nat[1]
...
```

Type equivalence:

Making It Dependent

The next target would be LF; this requires two ingredients:

0-usage:

```
let x: Nat[1] := 0 in
let p: x = 0 := eq_refl in                          // x: Nat[0]
let x' := x ← 1 in                                  // x: Nat[1]
...
```

```
forall A A' i j, A' = exchange A[i] A[j] → A[i] = A'[j]
```

Type equivalence:

Making It Dependent

The next target would be LF; this requires two ingredients:

0-usage:

```
let x: Nat[1] := 0 in
let p: x = 0 := eq_refl in                         // x: Nat[0]
let x' := x ← 1 in                                 // x: Nat[1]
...
```

```
forall A A' i j, A' = exchange A[i] A[j] → A[i] = A'[j]
```

Type equivalence:
It should be possible to use some off the shelf algorithm, using the functional semantics.

Generalizing Mutation

What if one wants to mutate $\langle\langle\rangle, \langle 0,1\rangle\rangle$ into $\langle\langle\rangle, \langle 10,1\rangle\rangle$?

Generalizing Mutation

What if one wants to mutate $\langle\langle\rangle, \langle 0,1\rangle\rangle$ into $\langle\langle\rangle, \langle 10,1\rangle\rangle$?

Direct style:

Introduce the notion of path/location of a type.

Generalizing Mutation

What if one wants to mutate ⟨⟨⟩, ⟨0,1⟩⟩ into ⟨⟨⟩, ⟨10,1⟩⟩?

Direct style:

Introduce the notion of path/location of a type.

```
let x := ⟨⟨⟩, ⟨0,1⟩⟩ in
let x' := x.2.1 ← 10 in                                   // Or x[1][0], etc...
...
```

Generalizing Mutation

What if one wants to mutate ⟨⟨⟩, ⟨0,1⟩⟩ into ⟨⟨⟩, ⟨10,1⟩⟩?

Direct style:

Introduce the notion of path/location of a type.

Generalizing Mutation

What if one wants to mutate ⟨⟨⟩, ⟨0,1⟩⟩ into ⟨⟨⟩, ⟨10,1⟩⟩?

Direct style:

Introduce the notion of path/location of a type.

FP²-inspired style:

Bind de-allocated memory locations for re-use.

A. Lorenzen, D. Leijen, W. Swierstra. 2023. FP²: Fully in-Place Functional Programming. ICFP (2023).

Generalizing Mutation

What if one wants to mutate ⟨⟨⟩, ⟨0,1⟩⟩ into ⟨⟨⟩, ⟨10,1⟩⟩?

Direct style:

Introduce the notion of path/location of a type.

FP²-inspired style:

Bind de-allocated memory locations for re-use.

```
let x := ⟨⟨⟩, ⟨0,1⟩⟩ in
let l1#⟨a,y⟩ := x in
let l2#⟨b,c⟩ := y in
let x' := l1#⟨a, l2#⟨b ← 10, c⟩⟩ in
...
```

A. Lorenzen, D. Leijen, W. Swierstra. 2023. FP²: Fully in-Place Functional Programming. ICFP (2023).

Generalizing Mutation

What if one wants to mutate ⟨⟨⟩, ⟨0,1⟩⟩ into ⟨⟨⟩, ⟨10,1⟩⟩?

Direct style:

Introduce the notion of path/location of a type.

FP^2-inspired style:

Bind de-allocated memory locations for re-use.

```
let x: Int[1] + Int[1] := Left 10
let x' := match x with
          | l#Left v ->  l#Right v
          | l#Right v -> l#Left v
in ...
```

A. Lorenzen, D. Leijen, W. Swierstra. 2023. FP^2: Fully in-Place Functional Programming. ICFP (2023).

Circling back: Quicksort

```
let Quicksort := λ x,
  let ⟨A, y⟩ := x in
  let ⟨p, r⟩ := y in
  if p < r
    let ⟨A, q⟩ := Partition ⟨A, ⟨p, r⟩⟩ in
    let A := Quicksort ⟨A, ⟨p, q-1⟩⟩ in
    Quicksort ⟨A, ⟨q+1, r⟩⟩
  else
    A
```

```
let Partition := λ x,
  let ⟨A, y⟩ := x in
  let ⟨p, r⟩ := y in
  let i := p-1
  let ⟨i, A⟩ = for j = p to r-1
    letr b := A[j] ≤ A[r] in
    if b
      let i := i+1 in
      let A := (swap i, j in A) in
      yield ⟨i, A⟩
    else
      yield ⟨i, A⟩
  in
  let A := (swap i, j in A) in
  return A, i+1
```

Circling back: Quicksort

```
let Quicksort := λ x,
  let ⟨A, y⟩ := x in
  let ⟨p, r⟩ := y in
  if p < r
    let ⟨A, q⟩ := Partition ⟨A, ⟨p, r⟩⟩ in
    let A := Quicksort ⟨A, ⟨p, q-1⟩⟩ in
    Quicksort ⟨A, ⟨q+1, r⟩⟩
  else
    A

let QS_sorted: forall A p r,
    let A' := Quicksort ⟨A, ⟨p, r⟩⟩ in
    forall x y,
    p ≤ x → x ≤ y → y < q →
    A'[x] ≤ A'[y]
```

```
let Partition := λ x,
  let ⟨A, y⟩ := x in
  let ⟨p, r⟩ := y in
  let i := p-1
  let ⟨i, A⟩ = for j = p to r-1
    letr b := A[j] ≤ A[r] in
    if b
      let i := i+1 in
      let A := (swap i, j in A) in
      yield ⟨i, A⟩
    else
      yield ⟨i, A⟩
  in
  let A := (swap i, j in A) in
  return A, i+1
```