‘do’ Unchained

Embracing Local Imperativity in a Purely Functional Language (Functional Pearl)

__Sebastian Ullrich__ (KIT), Leonardo de Moura (MSR)

Karlsruher Institut für Technologie

Microsoft Research

‘do’ Unchained

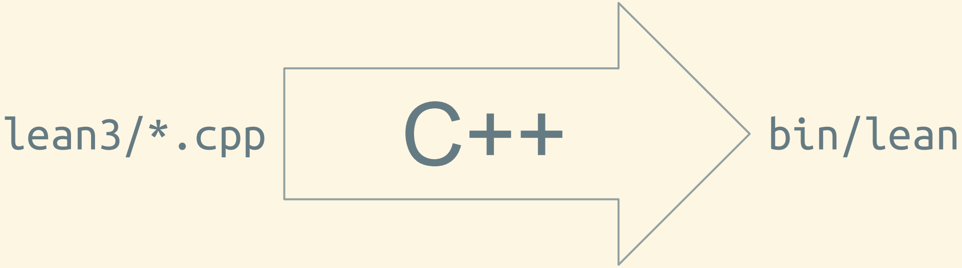
Embracing Local Imperativity in a Purely Functional Language (Functional Pearl)

(Lean)

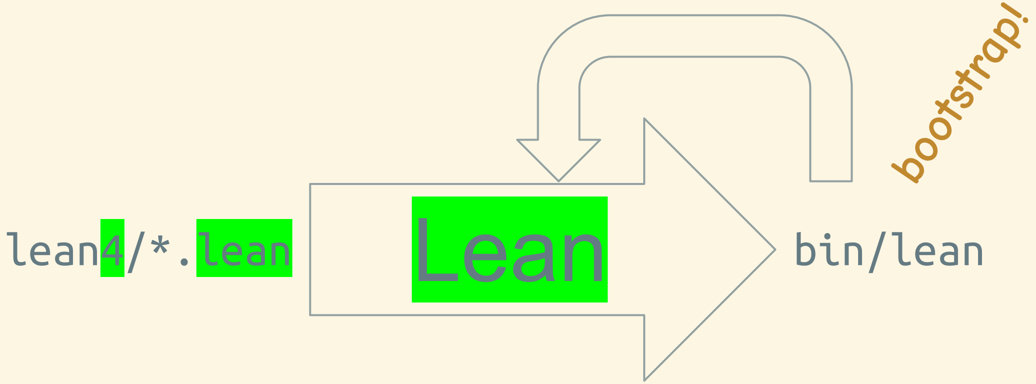
Sebastian Ullrich (KIT), *Leonardo de Moura* (MSR)

Karlsruher Institut für Technologie

The Lean 4 Project: Reimplementing Lean in Lean



The Lean 4 Project: Reimplementing Lean in Lean



The Grass is Greener on the Functional Side…?

Rust

```
let mut a_var = …;
let mut another_var = …;
if b1 {
  return a_var;
}
if b2 {
  a_var = f(a_var);
  another_var = g(a_var, another_var);
}
…
```

The Grass is Greener on the Functional Side…?

Rust

```
let mut a_var = …;
let mut another_var = …;
if b1 {
  return a_var;
}
if b2 {
  a_var = f(a_var);
  another_var = g(a_var, another_var);
}
…
```

Functional Lean

```
do let aVar := …
   let anotherVar := …
   if b1 then
     pure aVar
   else do
     let (aVar, anotherVar) ← if b2 then do
       let aVar ← f aVar
       let anotherVar ← g aVar anotherVar
       pure (aVar, anotherVar)
     else pure (aVar, anotherVar)
     …
```

The Grass is Greener on the Functional Side…?

Rust

```
let mut a_var = …;
let mut another_var = …;
if b1 {
  return a_var;
}
if b2 {
  a_var = f(a_var);
  another_var = g(a_var, another_var);
}
…
```

Imperative Lean

```
do let mut aVar := …
   let mut anotherVar := …
   if b1 then
     return aVar
   if b2 then
     aVar ← f aVar
     anotherVar ← g aVar anotherVar
   …
```

Semantics

Mutation… in My Theorem Prover?

New syntax translated modularly to **pure** code using monad transformers

- `let mut x : A := …` ⇒ `StateT A`
- `return (e : A)` ⇒ `ExceptT A`
- `for in/break/continue` ⇒ `ExceptT + fold + ExceptT`

See supplement for verification examples in Lean

Semantics, Level I

$$
\begin{aligned}
R(\texttt{return}\ e) &= \texttt{throw}\ e \\
R(e) &= \texttt{ExceptT.lift}\ e \\
R(\texttt{let}\ x \leftarrow s;\ s') &= \texttt{let}\ x \leftarrow R(s);\ R(s')
\end{aligned}
$$

Semantics, Level II

$$R(\texttt{return}\ \ e) \quad = \quad \texttt{throw}\ \ e$$
$$R(e) \quad = \quad \texttt{ExceptT.lift}\ \ e$$
$$R(\texttt{let}\ \ x \leftarrow\ \ s;\ \ s') \quad = \quad \texttt{let}\ \ x \leftarrow\ \ R(s);\ \ R(s')$$

```
syntax "return" term : stmt        syntax "return" : expander
macro_rules
  | `(stmt| expand! return in return $e) => `(stmt| throw $e)
  | `(stmt| expand! return in $e:term)   => `(stmt| ExceptT.lift $e)
  -- other rules subsumed by default `expand!` rules
```

Semantics, Level III

$$
\begin{aligned}
R(\texttt{return}\ \ e) &= \texttt{throw}\ \ e \\
R(e) &= \texttt{ExceptT.lift}\ \ e \\
R(\texttt{let}\ \ x \leftarrow\ \ s;\ \ s') &= \texttt{let}\ \ x \leftarrow\ R(s);\ \ R(s')
\end{aligned}
$$

$$
\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ \ e, \sigma \rangle \Rightarrow \langle \texttt{return}\ \ v, \sigma \rangle}
$$

Semantics, Level III

$$
\begin{array}{lcl}
R(\texttt{return}\ \ e) & = & \texttt{throw}\ \ e \\
R(e) & = & \texttt{ExceptT.lift}\ \ e \\
R(\texttt{let}\ \ x \leftarrow\ \ s;\ \ s') & = & \texttt{let}\ \ x \leftarrow\ \ R(s);\ \ R(s')
\end{array}
$$

$$
\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ \ e, \sigma \rangle \Rightarrow \langle \texttt{return}\ \ v, \sigma \rangle}
$$

$$
\frac{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ \ x \leftarrow\ \ s;\ \ s', \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}
$$

Semantics, Level III

$$R(\texttt{return}\ e) \quad = \quad \texttt{throw}\ e$$

$$R(e) \quad = \quad \texttt{ExceptT.lift}\ e$$

$$R(\texttt{let}\ x \leftarrow s;\ s') \quad = \quad \texttt{let}\ x \leftarrow R(s);\ R(s')$$

$$\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ e, \sigma \rangle \Rightarrow \langle \texttt{return}\ v, \sigma \rangle}$$

$$\frac{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ x \leftarrow s;\ s', \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \mid \Delta \vdash_f \texttt{return}\ e : m\ \alpha \hookrightarrow \omega}$$

immutable context

mutable context

inside of loop?

stmt. result type

block result type

Semantics, Level IV

$$\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ \ e, \sigma\rangle \Rightarrow \langle \texttt{return}\ \ v, \sigma\rangle}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \mid \Delta \vdash_f \texttt{return}\ \ e : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\langle s, \sigma\rangle \Rightarrow \langle n, \sigma'\rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ \ x \leftarrow\ \ s;\ \ s', \sigma\rangle \Rightarrow \langle n, \sigma'\rangle}$$

Semantics, Level IV

$$\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ \ e, \sigma\rangle \Rightarrow \langle \texttt{return}\ \ v, \sigma\rangle}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{return}\ \ e : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\langle s, \sigma\rangle \Rightarrow \langle n, \sigma'\rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ \ x \leftarrow\ \ s;\ \ s', \sigma\rangle \Rightarrow \langle n, \sigma'\rangle}$$

```
-- intrinsically typed definitional interpreter
def Do.eval [Monad m] : Do m α → m α
```

Semantics, Level IV

$$\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ \ e, \sigma\rangle \Rightarrow \langle \texttt{return}\ \ v, \sigma\rangle}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{return}\ \ e : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\langle s, \sigma\rangle \Rightarrow \langle n, \sigma'\rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ \ x \leftarrow\ \ s;\ \ s', \sigma\rangle \Rightarrow \langle n, \sigma'\rangle}$$

```
-- intrinsically typed definitional interpreter
def Do.eval [Monad m] : Do m α → m α


def R [Monad m] : Stmt Γ Δ f m α ω → Stmt Γ Δ f (ExceptT ω m) α Empty
```

Semantics, Level IV

$$\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ \ e, \sigma\rangle \Rightarrow \langle \texttt{return}\ \ v, \sigma\rangle}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{return}\ \ e : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\langle s, \sigma\rangle \Rightarrow \langle n, \sigma'\rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ \ x \leftarrow\ \ s;\ \ s', \sigma\rangle \Rightarrow \langle n, \sigma'\rangle}$$

```
-- intrinsically typed definitional interpreter
def Do.eval [Monad m] : Do m α → m α


def R [Monad m] : Stmt Γ Δ f m α ω → Stmt Γ Δ f (ExceptT ω m) α Empty


theorem trans_eq_eval [Monad m] [LawfulMonad m] : ∀ s : Do m α, Do.trans s = Do.eval s
```

Consequences

But What about Data Structures?

Lean's reference-counted runtime allows for *invisible destructive updates* [Ullrich, de Moura, 2019]

```
do let mut vec := #[]
   for x in xs do
     vec := vec.push (f x)  -- amortized O(1)
```

syntax & runtime semantics combine to a *Pure Imperative Programming* paradigm

A Bit of Evaluation

- extensively used in the implementation of Lean 4
- used in 31 out of 43 Lean 4 GitHub repositories
- even with the identity monad: `Id.run do …`

A Bit of Evaluation

- extensively used in the implementation of Lean 4
- used in 31 out of 43 Lean 4 GitHub repositories
- even with the identity monad: `Id.run do …`

```
example : StateM Nat Nat := do
  return 0
```

```
StateM Nat Nat
```

`return e` inside of a `do` block makes the surrounding block evaluate to `pure e`, skipping any further statements. Note that uses of the `do` keyword in other syntax like in `for _ in _ do` do not constitute a surrounding block in this sense; in supported editors, the corresponding `do` keyword of the surrounding block is

Conclusion

A new paradigm of Lean programming

An imperative embedding still amenable to verification

A case study in modular design, implementation & formalization of a syntax extension

Bibliography

Ullrich and de Moura: Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming, IFL'19.

Embedding

```
inductive Stmt (m : Type → Type u) (ω : Type) : (Γ Δ : List Type) → (b c : Bool) → (α : Type) → Type _ where
 | expr (e : Γ ⊢ Δ ⊢ m α) : Stmt m ω Γ Δ b c α
 | bind (s : Stmt m ω Γ Δ b c α) (s' : Stmt m ω (α :: Γ) Δ b c β) : Stmt m ω Γ Δ b c β  -- let _ ← s; s'
 | letmut (e : Γ ⊢ Δ ⊢ α) (s : Stmt m ω Γ (α :: Δ) b c β) : Stmt m ω Γ Δ b c β  -- let mut _ := e; s
 | assg (x : Fin Δ.length) (e : Γ ⊢ Δ ⊢ Δ.get x) : Stmt m ω Γ Δ b c Unit  -- x := e
 | ite (e : Γ ⊢ Δ ⊢ Bool) (s₁ s₂ : Stmt m ω Γ Δ b c α) : Stmt m ω Γ Δ b c α  -- if e then s₁ else s₂
 | ret (e : Γ ⊢ Δ ⊢ ω) : Stmt m ω Γ Δ b c α   -- return e
 | sfor (e : Γ ⊢ Δ ⊢ List α) (s : Stmt m ω (α :: Γ) Δ true true Unit) : Stmt m ω Γ Δ b c Unit  -- for _ in e do s
 | sbreak : Stmt m ω Γ Δ true c α  -- break
 | scont : Stmt m ω Γ Δ b true α  -- continue
```

Interpreter

```
def Stmt.eval [Monad m] (ρ : Assg Γ) : Stmt m ω Γ Δ b c α → Assg Δ → m (Neut ω α b c × Assg Δ)
 | expr e, σ => e[ρ][σ] >>= fun v => pure ⟨v, σ⟩
 | bind s s', σ =>
   let rec @[simp] cont val
     | ⟨Neut.val v, σ'⟩ => val v σ'
     | ⟨Neut.ret o, σ'⟩ => pure ⟨Neut.ret o, σ'⟩
     | ⟨Neut.rbreak, σ'⟩ => pure ⟨Neut.rbreak, σ'⟩
     | ⟨Neut.rcont, σ'⟩ => pure ⟨Neut.rcont, σ'⟩
   s.eval ρ σ >>= cont (fun v σ' => s'.eval (v :: ρ) σ')
 | letmut e s, σ =>
   s.eval ρ (e[ρ][σ], σ) >>= fun ⟨r, σ'⟩ => pure ⟨r, σ'.drop⟩
 | assg x e, σ => pure ⟨(), σ[x ↦ e[ρ][σ]]⟩
 | ite e s₁ s₂, σ => if e[ρ][σ] then s₁.eval ρ σ else s₂.eval ρ σ
 | ret e, σ => pure ⟨Neut.ret e[ρ][σ], σ⟩
 | sfor e s, σ =>
   let rec @[simp] go σ
     | [] => pure ⟨(), σ⟩
     | a::as =>
       s.eval (a :: ρ) σ >>= fun
       | ⟨(), σ'⟩ => go σ' as
       | ⟨Neut.rcont, σ'⟩ => go σ' as
       | ⟨Neut.rbreak, σ'⟩ => pure ⟨(), σ'⟩
       | ⟨Neut.ret o, σ'⟩ => pure ⟨Neut.ret o, σ'⟩
   go σ e[ρ][σ]
 | sbreak, σ => pure ⟨Neut.rbreak, σ⟩
 | scont, σ => pure ⟨Neut.rcont, σ⟩
termination_by
 eval s _ => (sizeOf s, 0)
 eval.go as => (sizeOf s, as.length)
```

A Bit of the Semantics

$$
\begin{array}{lll}
v \in \mathit{Val} & ::= \textbf{fun}\ x \ \texttt{=>}\ e \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid \texttt{cons}\ v_1\ v_2 \mid \dots & \subseteq \mathit{Expr} \\
n \in \mathit{Neut} & ::= v \mid \textbf{return}\ v \mid \textbf{break} \mid \textbf{continue} & \subseteq \mathit{Stmt} \\
\sigma \in \mathit{State} & \equiv \mathit{Var} \rightharpoonup \mathit{Val} &
\end{array}
$$

$$\boxed{e \Rightarrow v}$$

$$\dots \qquad \frac{\langle s, \emptyset \rangle \Rightarrow \langle n, \emptyset \rangle \qquad n \in \{v, \textbf{return}\ v\}}{\textbf{do}\ s \Rightarrow v}$$

$$\boxed{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}$$

$$\frac{e[\sigma] \Rightarrow v}{\langle e, \sigma \rangle \Rightarrow \langle v, \sigma \rangle} \quad \frac{\langle s, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle \qquad \langle s'[v/x], \sigma' \rangle \Rightarrow \langle n, \sigma'' \rangle}{\langle \textbf{let}\ x \leftarrow s;\ s', \sigma \rangle \Rightarrow \langle n, \sigma'' \rangle} \quad \frac{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle \qquad n \notin \mathit{Val}}{\langle \textbf{let}\ x \leftarrow s;\ s', \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}$$

$$\frac{x \notin \sigma \qquad e[\sigma] \Rightarrow v \qquad \langle s, \sigma[x \mapsto v] \rangle \Rightarrow \langle n, \sigma' \rangle}{\langle \textbf{let mut}\ x\ \texttt{:=}\ e;\ s, \sigma \rangle \Rightarrow \langle n, \sigma'[x \mapsto \bot] \rangle} \quad \frac{x \in \sigma \qquad e[\sigma] \Rightarrow v}{\langle x\ \texttt{:=}\ e;\ s, \sigma \rangle \Rightarrow \langle (), \sigma[x \mapsto v] \rangle}$$

$$\frac{e[\sigma] \Rightarrow v}{\langle \textbf{return}\ e, \sigma \rangle \Rightarrow \langle \textbf{return}\ v, \sigma \rangle} \quad \frac{e[\sigma] \Rightarrow \texttt{nil}}{\langle \textbf{for}\ x\ \textbf{in}\ e\ \textbf{do}\ s, \sigma \rangle \Rightarrow \langle (), \sigma \rangle}$$

Type System

$$\boxed{\Gamma \vdash e : \tau}$$

$$\ldots \quad \frac{\Gamma \mid \cdot \vdash_{\texttt{nofor}} s : m\ \alpha \hookrightarrow \alpha}{\Gamma \vdash \textbf{do}\ \ s : m\ \alpha} \qquad \frac{}{\Gamma \vdash () : \mathsf{Unit}} \qquad \frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{List}\ \alpha} \qquad \frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash e' : \mathsf{List}\ \alpha}{\Gamma \vdash \mathsf{cons}\ \ e\ \ e' : \mathsf{List}\ \alpha}$$

$$\boxed{\Gamma \mid \Delta \vdash_f s : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta \vdash e : m\ \alpha}{\Gamma \mid \Delta \vdash_f e : m\ \alpha \hookrightarrow \omega} \qquad \frac{x \notin \Delta \qquad \Gamma \mid \Delta \vdash_f s : m\ \alpha \hookrightarrow \omega \qquad \Gamma, x : \alpha \mid \Delta \vdash_f s' : m\ \beta \hookrightarrow \omega}{\Gamma \mid \Delta \vdash_f \textbf{let}\ \ x \leftarrow s;\ \ s' : m\ \beta \hookrightarrow \omega}$$

$$\frac{x \notin \Gamma, \Delta \qquad \Gamma, \Delta \vdash e : \alpha \qquad \Gamma \mid \Delta, x : \alpha \vdash_f s : m\ \beta \hookrightarrow \omega}{\Gamma \mid \Delta \vdash_f \textbf{let mut}\ \ x\ := e;\ \ s : m\ \beta \hookrightarrow \omega} \qquad \frac{\Gamma, \Delta, x : \alpha \vdash e : \alpha}{\Gamma \mid \Delta, x : \alpha \vdash_f x\ := e : m\ \mathsf{Unit} \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta \vdash e : \mathsf{Bool} \qquad \Gamma \mid \Delta \vdash_f s_1 : m\ \alpha \hookrightarrow \omega \qquad \Gamma \mid \Delta \vdash_f s_2 : m\ \alpha \hookrightarrow \omega}{\Gamma \mid \Delta \vdash_f \textbf{if}\ \ e\ \ \textbf{then}\ \ s_1\ \ \textbf{else}\ \ s_2 : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \mid \Delta \vdash_f \textbf{return}\ \ e : m\ \alpha \hookrightarrow \omega} \qquad \frac{x \notin \Delta \qquad \Gamma, \Delta \vdash e : \mathsf{List}\ \alpha \qquad \Gamma, x : \alpha \mid \Delta \vdash_{\texttt{for}} s : m\ \mathsf{Unit} \hookrightarrow \omega}{\Gamma \mid \Delta \vdash_f \textbf{for}\ \ x\ \ \textbf{in}\ \ e\ \ \textbf{do}\ \ s : m\ \mathsf{Unit} \hookrightarrow \omega}$$

$$\frac{}{\Gamma \mid \Delta \vdash_{\texttt{for}} \textbf{break} : m\ \alpha \hookrightarrow \omega} \qquad \frac{}{\Gamma \mid \Delta \vdash_{\texttt{for}} \textbf{continue} : m\ \alpha \hookrightarrow \omega}$$