‘do’ Unchained: Embracing Local Imperativity in a Purely Functional Language (Functional Pearl)

SEBASTIAN ULLRICH, Karlsruhe Institute of Technology, Germany
LEONARDO DE MOURA, Microsoft Research, USA

Purely functional programming languages pride themselves with reifying effects that are implicit in imperative languages into reusable and composable abstractions such as monads. This reification allows for more exact control over effects as well as the introduction of new or derived effects. However, despite libraries of more and more powerful abstractions over effectful operations being developed, syntactically the common **do** notation still lags behind equivalent imperative code it is supposed to mimic regarding verbosity and code duplication. In this paper, we explore extending **do** notation with other imperative language features that can be added to simplify monadic code: local mutation, early return, and iteration. We present formal translation rules that compile these features back down to purely functional code, show that the generated code can still be reasoned over using an implementation of the translation in the Lean 4 theorem prover, and formally prove the correctness of the translation rules relative to a simple static and dynamic semantics in Lean.

CCS Concepts: • **Software and its engineering** → **Language features**.

Additional Key Words and Phrases: functional programming, interactive theorem proving, Lean

ACM Reference Format:
Sebastian Ullrich and Leonardo de Moura. 2022. ‘do’ Unchained: Embracing Local Imperativity in a Purely Functional Language (Functional Pearl). *Proc. ACM Program. Lang.* 6, ICFP, Article 109 (August 2022), 28 pages. https://doi.org/10.1145/3547640

1 INTRODUCTION

The success story of Haskell’s perhaps most well-known abstraction, the monad as popularized by Wadler [1990], is by now invariably linked with its ubiquitous syntax sugar, the **do** notation. Together they can express imperative sequencing on a more general, well-behaved abstraction level while retaining a terse, familiar, and suggestive syntax. They are in fact so commonly linked, and taught, together that programmers tend to use them even when weaker, potentially more performant abstractions would suffice [Marlow et al. 2016].

It is then surprising that no serious attempts to add more imperative control flow techniques over just sequencing seem to have been made, unless one considers that most of these only carry their weight in the presence of mutable variables. For example, there is not much reason to introduce an **if _ then _** syntax without an **else** branch to Haskell when there is already the when combinator that can be used to the same effect. Mutable variables in turn are of course seen as an antithesis to purely functional programming’s core tenet of referential transparency.

On the other hand, even imperative languages have started to rein in mutability and make it the exception, not the rule. In Rust [Matsakis and Klock 2014], for example, **let**-bound variables

Authors’ addresses: Sebastian Ullrich, sebastian.ullrich@kit.edu, Karlsruhe Institute of Technology, Kaiserstraße 12, Karlsruhe, 76131, Germany; Leonardo de Moura, leonardo@microsoft.com, Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA.

2475-1421/2022/8-ART109
https://doi.org/10.1145/3547640

are immutable unless explicitly marked **mut**, and mutable references are non-shareable to prevent “spooky actions at a distance”. Ultimately, mutable variables in imperative languages (excluding mutable references) are in fact commonly compiled down to static single assignment form [Rosen et al. 1988], which is known to be equivalent to a subset of continuation passing style [Kelsey 1995], meaning it is certainly possible to transform them to pure code.

Starting with mutable variables, we thus explore embracing imperative language features as part of the **do** notation in this paper. We do so in the context of the Lean 4 programming language and theorem prover [de Moura and Ullrich 2021], though the implementation can readily be adapted to any other functional language with support for monads and monad transformers. We will consider three extensions in total, which at this point we merely want to tease using suggestive side-by-side examples of Rust and Lean code:

- mutable variables (Section 2),

```
let mut x = read_int();
if x != 0 {
  x = f(x);
}
...
```

```
do let mut x ← readInt
   if x != 0 then
     x := f x
   ...
```

- early return (Section 3),

```
let line = read_line();
if line == "" {
  return
}
...
```

```
do let line ← readLine
   if line = "" then
     return ()
   ...
```

- and **for** loops (Section 4).

```
let lines = read_lines();
let mut sum = 0;
for line in lines {
  if (line == "END") {
    break
  }
  sum += parse_int(line);
}
...
```

```
do let lines ← readLines
   let mut sum := 0
   for line in lines do
     if line = "END" then
       break
     sum := sum + parseInt line
   ...
```

In each section, we will motivate the extension using examples, discuss possible syntax and semantics, focusing on those that are least surprising to both imperative and functional programmers, and give a formal translation into purely functional code. We validate the translation with an incremental, one-to-one implementation in the paper’s supplemental material [Ullrich and de Moura 2022] using Lean’s flexible macro system. We discuss this reference implementation as well as a more extensive implementation built into Lean itself (Section 5). We show that the code produced by our translation can still be analysed and reasoned over using the same proof construction tools we use for pure code (Section 6), and formally prove in Lean that the translation preserves the input’s static and dynamic semantics specified as a simple type system and natural semantics (Section 7). We conclude with an evaluation of the use of the extensions in our own code as well as that of third parties (Section 8).

Key contributions.

- We extend the **do** DSL with support for mutable variables as well as the familiar imperative statements **return**, **for**, **break**, and **continue**, usable in both monadic and pure computations.
- We give a formal translation of these extensions, backed by a reference implementation and a formal equivalence proof in Lean, down to well-known combinators so that the core language does not have to be changed.
- Throughout, we discuss issues of syntax, semantics, compilation, and proving.

2 LOCAL MUTATION

One incentive for introducing mutable variables to Lean was the proliferation of the following "conditional update" pattern in the Lean codebase:[1]

```
do ...
   x ← if b then
       f x
     else pure x
   ...
```

We cannot help but notice that the **else** branch feels redundant: nothing of interest is happening in it. While the boilerplate in this abstract case is still minimal, the issue compounds in the presence of multiple variables with names of more realistic length.

```
do ...
   (aVar, anotherVar) ← if someCondition then do
       aVar ← transform1 aVar
       anotherVar ← transform2 aVar anotherVar
       pure (aVar, anotherVar)
     else pure (aVar, anotherVar)
   ...
```

This code is still easy enough to scan and comprehend, but we would be hard-pressed to defend it to e.g. a programmer coming from Rust, who might expect something more like

```
do let mut aVar ...
   ...
   if someCondition then
     aVar ← transform1 aVar
     anotherVar ← transform2 aVar anotherVar
   ...
```

No matter how much we extoll the virtues of purely functional code to the programmer, it is unlikely we can persuade them, or ourselves, that the latter code is not easier to read, comprehend, and ultimately maintain. At the same time, the boilerplate is mostly about bindings and not data so that we cannot profitably abstract it into a new (higher-order) combinator. Thus we finally relent and instead perform the before-unthinkable: we will try to assign the above code semantics that are reasonable in the eyes of both functional and imperative programmers.

We start by copying the careful approach of Rust and other languages of introducing separate binding syntax for mutable variables: `let mut x := ...`[2], and `let mut x ← ...` for the monadic form.

[1] We note that monadic bindings in both Haskell and Lean are non-recursive, thus shadowing works as expected even if the right-hand side contains a reference to the shadowed variable.

[2] In Lean, we usually use := where Haskell would use = since the latter is instead used for the equality type.

This way, we can make sure that users will not accidentally make use of local mutation without even knowing it exists. For a thus introduced variable `x`, `x := ...` seems like the obvious choice of syntax for reassigning the variable to the value of a pure term. Unfortunately, the corresponding syntax with a monadic ← is already taken. Since distinguishing between declaration and reassignment is certainly a good idea, for Lean 4 we have decided on the drastic step of realigning the monadic binding syntax to the pure one and changing the syntax for introducing an immutable binding to **`let`** `x ← ...` as seen in the examples in Section 1.[3] Thus a variable definition in Lean 4 is uniformly signified by the **`let`** keyword.

With syntactic questions out of the way, let us now turn to the expected semantics, starting with the question of scope. Declaring a mutable variable should grant us access to its value in the subsequent code just like with immutable ones, but reassignment will have to be more limited: if there is another **do** block nested anywhere inside the first one, say within an argument to some monadic combinator, there is no way in general to propagate reassignments back to the outer block without true mutation. Neither changing the result type nor introducing a new monadic layer to do the propagation is guaranteed to work (or even typecheck) in all contexts of the inner block. Thus we will sensibly restrict reassignment of a variable to the same **do** block it was declared in using **`let mut`**. This also resolves the question as to how mutable variables should behave when the block is executed multiple times: in any single execution, the variable is first freshly declared and then mutated, so the executions are independent. However, a similar but more subtle problem exists for some monads, those that may execute the >>= right-hand side more than once, such as the list/nondeterminism and the continuation monad:

```
do let mut x := 0
   let y ← choose [0, 1, 2, 3]
   x := x + 1
   guard (x < 3)
   pure (x + y)
```

In usual implementations of nondeterminism, this program will bind `y` to the values `0`, . . . , `3` in turn, execute the remainder of the block with each of them, and then collect all the results from the different executions in a list. The `guard` function discards the "strands" of execution for which the given condition is false. Should the reassignment of `x` then persist into further executions, creating the output `[1, 3]`, or should it be limited to the current execution only, yielding `[1, 2, 3, 4]`? We argue that the more intuitive (and implementable) semantics is the latter one, where local mutation is interpreted as a *local state effect on top* of the underlying monad. Thus re-running a nondeterministic program or captured continuation is still "pure": mutable variables will start out with the same values as in the first run. With the alternative semantics, it would not even be the case that our initial examples from the beginning of the section are indeed equivalent in all monads. If we wanted to implement these "impure" semantics, we would to introduce the state effect *below* the nondeterminism effect (or rely on existing arbitrary state from the base monad such as `IO` or `ST`), which is not an option in general as not all monad transformers commute.

We start the discussion of the formal translation of these semantics with a basic syntax and desugaring of **do** similar to that in the Haskell 98 report [Jones 2003] (Figure 1). Here **do** is followed by one of two kinds of *statements*, which are defined inductively: either a plain expression, or a monadic binding followed by another statement. The value of an expression as a statement is that of the expression (D1), while a monadic binding desugars to an application of the monadic bind operator >>= (D2). We do not directly rewrite occurrences of **do** into other **do** terms but do so using a recursive helper function D, which will become useful in Section 3. For the sake of presentation, we

[3] In fact, the "new" syntax is reminiscent of the original monadic binding notation (let $x \Leftarrow e$ in e') in Moggi [1991].

Syntax

$$x, y \in \mathit{Var}$$

$$\begin{aligned} e \in \mathit{Expr} &::= \texttt{do}\ s \\ &\quad \mid x \mid \texttt{fun}\ x\ \texttt{=>}\ e \mid \texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ e' \mid e\ e' \mid e\ \texttt{>>=}\ e' \mid \ldots \\ s \in \mathit{Stmt} &::= e \\ &\quad \mid \texttt{let}\ x \leftarrow s;\ s' \end{aligned}$$

Translation

$$\texttt{do}\ s \rightsquigarrow D(s) \tag{1}$$

$$D : \mathit{Stmt} \longrightarrow \mathit{Expr}$$

$$D(e) = e \tag{D1}$$

$$D(\texttt{let}\ x \leftarrow s;\ s') = D(s)\ \texttt{>>=}\ \texttt{fun}\ x\ \texttt{=>}\ D(s') \tag{D2}$$

Abbreviations

$$\texttt{let}\ x\ \texttt{:=}\ e;\ s \equiv \texttt{let}\ x \leftarrow \texttt{pure}\ e;\ s \tag{A1}$$

$$s;\ s\texttt{'} \equiv \texttt{let}\ \texttt{x} \leftarrow s;\ s\texttt{'} \tag{A2}$$

Fig. 1. A basic **do** translation

Operations

$$\texttt{pure} : \alpha \longrightarrow m\ \alpha$$

$$(\texttt{>>=}) : m\ \alpha \longrightarrow (\alpha \longrightarrow m\ \beta) \longrightarrow m\ \beta$$

Laws

$$\begin{aligned} (x\ \texttt{>>=}\ f)\ \texttt{>>=}\ g &= x\ \texttt{>>=}\ (\texttt{fun}\ a\ \texttt{=>}\ f\ a\ \texttt{>>=}\ g) \\ \texttt{pure}\ a\ \texttt{>>=}\ f &= f\ a \\ f\ \texttt{>>=}\ \texttt{pure} &= f \end{aligned}$$

Fig. 2. Basic monadic operations and their laws. `pure` is usually defined in the more general `Applicative` typeclass, which comes with even more operations and laws, but we will focus on monads for this paper.

also introduce a syntax for *pure* let bindings as an abbreviation for monadic bindings of applications of the `pure` function (A1). In the translation rules, we will assume that all abbreviations have already been unfolded. A more direct translation of pure let bindings is certainly possible[4], but equivalent in Lean under the standard monad laws (Figure 2), so we use this shortcut to minimize the number of translation cases that we need to consider. Similarly, we will restrict ourselves to variable bindings instead of more complex pattern bindings, but the translation naturally extends to the latter, as is done in the full implementation in Lean 4 (Section 5.3). Finally, we introduce statement sequencing $s;\ s\texttt{'}$ not as a primitive but as yet another abbreviation, in this case for a binding to a fresh variable name `x` (contrasted by its monospace font from *metavariables* in italics) (A2). For simplicity, we will assume in the following that the underlying rewriting system is *hygienic* as is the case for our

[4] and necessary in languages like Haskell where pure and monadic bindings have different shadowing/recursive semantics

Syntax

$$\begin{aligned} s \in \mathit{Stmt} ::= {} & \ldots \\ & \mid \texttt{let mut}\ x\ \texttt{:=}\ e;\ s \\ & \mid x\ \texttt{:=}\ e \\ & \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \end{aligned}$$

Abbreviations

$$\texttt{let mut}\ x \leftarrow s;\ s' \equiv \texttt{let}\ \texttt{y} \leftarrow s;\ \texttt{let mut}\ x\ \texttt{:=}\ \texttt{y};\ s' \tag{A3}$$

$$x \leftarrow s \equiv \texttt{let}\ \texttt{y} \leftarrow s;\ x\ \texttt{:=}\ \texttt{y} \tag{A4}$$

$$\texttt{if}\ e\ \texttt{then}\ s_1 \equiv \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ \texttt{pure ()} \tag{A5}$$

$$\texttt{unless}\ e\ \texttt{do}\ s_2 \equiv \texttt{if}\ e\ \texttt{then}\ \texttt{pure ()}\ \texttt{else}\ s_2 \tag{A6}$$

Fig. 3. Syntax of a **do** with local mutation

reference implementation (Section 5.1), meaning that such an identifier introduced by a rewriting rule is renamed automatically to avoid any conflict with user-specified names.

We note that the grammar as presented has been optimized for translation, not parsing, and is thus ambiguous regarding the associativity of semicolons. We will tacitly assume that they always associate to the right and will use parentheses where necessary, i.e. **let** $x \leftarrow s_1;\ s_2;\ s_3$ represents the same statement as **let** $x \leftarrow s_1;\ (s_2;\ s_3)$, not **let** $x \leftarrow (s_1;\ s_2);\ s_3$ or (**let** $x \leftarrow s_1;\ s_2);\ s_3$. In the reference implementation, we resolve the issue using parsing precedences and a curly braces notation for grouping statements (Section 5.1). The full implementation in Lean also supports a Haskell-inspired indentation-sensitive syntax as seen in Section 1, which we will use in examples.

The syntax from Figure 1 already diverges from that known from Haskell or previous versions of Lean in two distinct ways: firstly, we denote both kinds of bindings with a leading **let** keyword as discussed above. Secondly, our monadic binding binds not just a term but another statement. With the grammar at hand, this isn't very useful yet because nested bindings can always be floated out by the monad associativity law (Figure 2).

Nested bindings become a more interesting option when we add control flow statements, which we do together with adding syntax for local mutation in Figure 3: **let mut** x := $e;\ s$ introduces a mutable variable x that can later (inside s) be *reassigned* using x := e. We introduce monadic equivalents **let mut** $x \leftarrow s;\ s'$ and $x \leftarrow s$ by desugaring them to the sequence of a temporary, non-mutable monadic binding and the respective pure statement (A3, A4), thus simplifying our translation by keeping **let** $x \leftarrow s;\ s'$ as the only primitive monadic binding.

In straight-line code, local mutation is not very interesting: in **let mut** $x \leftarrow s;\ \ldots;\ x \leftarrow s';\ \ldots$, we could achieve the expected semantics by replacing both bindings with **let** $x \leftarrow \ldots$ and relying on shadowing for "mutation".

Thus we also introduce the conditional statement **if** e **then** s_1 **else** s_2 as well as derived syntax abbreviations (A5, A6) so local mutation becomes meaningful:

```
do let mut a ← f
   if b a then
     a ← g a
   pure a
```

By the informal semantics discussed above, we expect the translation of the above code block to be equivalent to the term `f >>= fun a => if b a then g a else pure a`. Let us start by unfolding all abbreviations we have introduced.

```
do let y ← f
   let mut a := y
   let x ←
     if b a then
       let y' ← g a
       a := y'
     else
       pure ()
   pure a
```

We can easily eliminate the mutable variable by passing the updated state outwards, much like we did manually in our very first example in this section:

```
do let y ← f
   let a := y
   let (x, a) ←
     if b a then
       let y' ← g a
       let a := y'
       pure ((), a)
     else
       pure ((), a)
   pure a
```

By the monad laws, this is in fact equivalent to the expected term. Like GHC [Peyton Jones 1996], the Lean compiler can optimize the code above by inlining `pure` and >>= (if known) and performing transformations on *match-of-match* (known as *case-of-case* in GHC) and *match-of-constructor*. For example, assuming the monad in the example above is `Reader`, Lean generates code which is equivalent to

```
fun r =>
  let a := f r
  match b a with
  | false => a
  | true  => g a r
```

Experienced purely functional programmers might notice a familiar pattern in the state-returning code: putting the “return value” and state into a pair and then extracting them at >>= mirrors the implementation of the `StateT` monad transformer (Figure 4). Indeed, we can rewrite the code using it:

```
do let y ← f
   let a := y
   StateT.run' (do
     let x ←
       if b a then
         let y' ← StateT.lift (g a)
         set y'
       else
         StateT.lift (pure ())
     let a ← get
     pure a) a
```

Operations

$$
\begin{array}{ll}
\texttt{StateT.run'} & : \texttt{StateT}\ \sigma\ m\ \alpha \rightarrow\ \sigma \rightarrow\ m\ \alpha \\
\texttt{StateT.lift} & : m\ \alpha\ \rightarrow \texttt{StateT}\ \sigma\ m\ \alpha \\
\texttt{get} & : \texttt{StateT}\ \sigma\ m\ \sigma \\
\texttt{set} & : \sigma\ \rightarrow \texttt{StateT}\ \sigma\ m\ \texttt{Unit}
\end{array}
$$

Properties

$$
\begin{array}{lcl}
\texttt{StateT.run' (pure}\ a\texttt{)}\ s & = & \texttt{pure}\ a \\
\texttt{StateT.run' (StateT.lift}\ x\ \texttt{>>=}\ f\texttt{)}\ s & = & x\ \texttt{>>=}\ \textbf{fun}\ a\ \texttt{=> StateT.run'}\ (f\ a)\ s \\
\texttt{StateT.run' (get >>=}\ f\texttt{)}\ s & = & \texttt{StateT.run'}\ (f\ s)\ s \\
\texttt{StateT.run' (set}\ s'\ \texttt{>>=}\ f\texttt{)}\ s & = & \texttt{StateT.run'}\ (f\ ())\ s'
\end{array}
$$

Fig. 4. `StateT` monad transformer operations and relevant properties for reasoning about them

Translation

$$
\begin{array}{lcll}
D(\textbf{let mut}\ x\ \texttt{:=}\ e\texttt{;}\ s) & = & \textbf{let}\ x\ \texttt{:=}\ e\texttt{; StateT.run'}\ D(S_x(s))\ x & \text{(D3)} \\
D(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2) & = & \textbf{if}\ e\ \textbf{then}\ D(s_1)\ \textbf{else}\ D(s_2) & \text{(D4)}
\end{array}
$$

$$
\begin{array}{lclll}
\multicolumn{5}{l}{S : \mathit{Var} \rightarrow \mathit{Stmt} \rightarrow \mathit{Stmt}} \\
S_y(e) & = & \texttt{StateT.lift}\ e & & \text{(S1)} \\
S_y(\textbf{let}\ x \leftarrow s\texttt{;}\ s') & = & \textbf{let}\ x \leftarrow S_y(s)\texttt{;}\ \textbf{let}\ y \leftarrow \texttt{get;}\ S_y(s') & \text{if } x \neq y & \text{(S2)} \\
S_y(\textbf{let mut}\ x\ \texttt{:=}\ e\texttt{;}\ s) & = & \textbf{let mut}\ x\ \texttt{:=}\ e\texttt{;}\ S_y(s) & \text{if } x \neq y & \text{(S3)} \\
S_y(x\ \texttt{:=}\ e) & = & x\ \texttt{:=}\ e & \text{if } x \neq y & \text{(S4)} \\
S_y(y\ \texttt{:=}\ e) & = & \texttt{set}\ e & & \text{(S5)} \\
S_y(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2) & = & \textbf{if}\ e\ \textbf{then}\ S_y(s_1)\ \textbf{else}\ S_y(s_2) & & \text{(S6)}
\end{array}
$$

Fig. 5. Translation of a **do** with local mutation

Here we introduce a *state effect* for the mutable variable a using `StateT.run'`, then `get` and `set` the current value inside. We remark that `set` corresponds to Haskell's `put` method for setting the state within the monad. We lift all existing monadic actions to the base monad using `StateT.lift` (which is a no-op for the specific case of `pure`).

For this simple example, the rewritten code is not exactly simpler than the previous version. However, the main motivation for abstracting translation of mutable variables into a separate effect is that it does not only simplify the presentation of that translation, but it will also ensure *modularity* of our extension with others defined similarly: by separating every extension into its own effect, we can layer them naturally without having to manually reconcile their interaction. In other words, while we might need to add new translation rules for local mutation when introducing new **do** syntax in later sections, we will not have to modify existing rules for them. We will not have to worry about how to preserve the state on **break**, nor about how **return** inside loops has to be handled.

Figure 5 gives a formal translation of mutable variables to state effects: when encountering the definition of a mutable variable y, we use the helper function S_y to lift the following statements, i.e.

y's scope, into the state transformer and rewrite them appropriately (D3). For monadic actions e, we do this with `StateT.lift` (S1). We use the shadowing approach for binding the current value of y, starting with its initial value in (D3), so we need to rebind it after reassignments as well as at possible control flow join points. In our reduced grammar, this can only be the case between the two statements in **let** $x \leftarrow s;\ s'$, so we add a binding **let** $y \leftarrow$ `get` in between them (S2). This step could of course be elided if there is no reassignment of y in s. Furthermore, it only makes sense if x and y are distinct variables, which we enforce. We also do so for **let mut** x, meaning shadowing of mutable variables is disallowed in general. Apart from the implementation, another reason for this restriction is to avoid any confusion on the user's side between mutable and immutable bindings. Finally, on a reassignment y `:=` e of the variable in question, we `set` the new value (S5).

$S_y(s)$ will thus eliminate any reassignments of y in s. If any remaining reassignments are encountered in the main translation function D, that must mean that no such mutable variable is in scope, and an appropriate error should be generated.

3 EARLY RETURN

Now that we have support for basic imperative control flow in **do**, it makes sense to talk about supporting **return** as well. While not without its own controversies, the programming pattern of *early return* seems to generally be well regarded in imperative programming for quickly discharging trivial/pathological cases in the beginning of a function without introducing indentation creep from nested conditionals. This matches our experience with it in Lean such as in the following example.

```
def isDefEqSingleton (structName : Name) (s : Expr) (v : Expr) : MetaM Bool := do
  let ctorVal ← getStructureCtor structName
  if ctorVal.numFields != 1 then
    return false  -- not a structure with a single field
  let s ← whnf s
  if !s.isMVar then
    return false  -- not an unsolved metavariable
  ...
```

This code taken from the Lean unifier and slightly simplified for presentation tries to reduce a unification problem `p s` $\overset{?}{=}$ `v` where `p` is the projection of a single-field structure type to `s` $\overset{?}{=}$ `c v` where `c` is the constructor of that type, using early return to quickly abort when the problem is not of the expected shape. As we shall see in the next section, **return** becomes even more useful when combined with iteration.

As with mutation, the reasonable semantics we can implement without changing code outside the **do** block is local: we will implement **return** e to abort execution *of the current* **do** *block* and have it return the value of e. We will discuss this decision more in Section 8.

Before we get to the implementation, a quick syntax consideration: in Lean 4, even before introducing the extended **do** block, we had already removed our analog of Haskell's `return` function in favor of the more general `pure`. Thus the decision to requisition the word as a keyword known from many imperative languages instead was a relatively easy one.

Programmatically, we could implement support for **return** by restructuring the entire **do** block into the noted nested conditionals. Even more so than in the previous section, however, it turns out that we can implement the desired semantics with relatively few additional rules by introducing a new effect (Figure 7), this time using the exception monad transformer `ExceptT` (Figure 6): we implement **return** e by raising e as an exception (R1), lifting all other monadic actions into the monad using `ExceptT.lift` (R2), and finally catching the exception, if any, and returning its captured

Operations

$$\begin{aligned}
&\texttt{runCatch} &&: \texttt{ExcepT}\ \alpha\ m\ \alpha \longrightarrow\ m\ \alpha \\
&\texttt{ExcepT.lift} &&: m\ \alpha \longrightarrow \texttt{ExcepT}\ \varepsilon\ m\ \alpha \\
&\texttt{throw} &&: \varepsilon \longrightarrow \texttt{ExcepT}\ \varepsilon\ m\ \alpha
\end{aligned}$$

Properties

$$\begin{aligned}
&\texttt{runCatch (pure}\ a) &&= \texttt{pure}\ a \\
&\texttt{runCatch (ExceptT.lift}\ x\ \texttt{>>=}\ f)\ s &&= x\ \texttt{>>=}\ \textbf{fun}\ a\ \texttt{=> runCatch}\ (f\ a) \\
&\texttt{runCatch (throw}\ a\ \texttt{>>=}\ f) &&= \texttt{pure}\ a
\end{aligned}$$

Fig. 6. ExceptT monad transformer operations and relevant properties for reasoning about them

Syntax

$$\begin{aligned}
s \in \mathit{Stmt} &::= \ldots \\
&\mid\ \textbf{return}\ e
\end{aligned}$$

Translation

$$\textbf{do}\ s \ \rightsquigarrow\ \texttt{runCatch}\ D(R(s)) \tag{1'}$$

$$R : \mathit{Stmt} \longrightarrow \mathit{Stmt}$$

$$\begin{aligned}
&R(\textbf{return}\ e) &&= \texttt{throw}\ e && \text{(R1)} \\
&R(e) &&= \texttt{ExceptT.lift}\ e && \text{(R2)} \\
&R(\textbf{let}\ x \leftarrow s;\ s') &&= \textbf{let}\ x \leftarrow R(s);\ R(s') && \text{(R3)} \\
&R(\textbf{let mut}\ x\ \texttt{:=}\ e;\ s) &&= \textbf{let mut}\ x\ \texttt{:=}\ e;\ R(s) && \text{(R4)} \\
&R(x\ \texttt{:=}\ e) &&= x\ \texttt{:=}\ e && \text{(R5)} \\
&R(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2) &&= \textbf{if}\ e\ \textbf{then}\ R(s_1)\ \textbf{else}\ R(s_2) && \text{(R6)}
\end{aligned}$$

Fig. 7. A **do** with early return via exceptions

value at the top level of the **do** block (1') using runCatch (Figure 6). This way, we get the short-circuiting semantics for free from ExceptT's implementation of >>= introduced by our unchanged translation of **let** $x \leftarrow s;\ s'$. The only existing rule we had to change was not that of an extension but the basic top-level translation rule (1). If a **do** block does not contain any **return** statements, we can of course fall back to the original rule instead. Note also that we did not have to extend S_y at all in this case because the only new syntax, **return** e, is eliminated before D, and therefore S_y, is ever run.

4 ITERATION

One of the first things a functional programmer usually learns is that loops from imperative languages can be replaced by recursion. However, the mere fact of equivalence does not imply that the translation is always as readable or maintainable as the original. One issue with recursive helper definitions is that of textual locality: we have to define the recursion either before or after its (usually singular) use site even if it by itself is not a self-contained abstraction, moving it out of its surrounding context and hurting code comprehension compared to in-place usage of loops.

Focusing first on iteration over the elements of a collection, perhaps the most common kind of loop, we note that the locality issue can be avoided by use of folds, which allow in-place iteration using anonymous functions. We illustrate this kind of iteration using lists

```
List.foldlM (f : δ → α → m δ) (init : δ) (xs : List α) : m δ
```

where `List α` is a list of elements of type α and δ is the accumulator type. In our experience in working on the Lean codebase, where fold-like traversal is pervasive, folds work relatively well for cases of simple control flow and one or two datums kept in the accumulator.

```
do ...
   let (x, y) ← zs.foldlM (init := (x, y)) $ fun (x, y) z => do
     let x' ← f x z
     if p x' then
       pure (x', g y z)
     else
       pure (x', y)
   ...
```

Lean’s support for named parameters, the `$` infix operator for function application, and extended dot notation avoids some confusion about parameter order as well as having to parenthesize the “loop body”. The term `zs.foldlM (init := a) f` is notation for `List.foldlM f a zs`. However, as soon as the number of “mutables” increases and/or the control flow inside the loop body gets more complex, handling and updating of the state tuple can quickly get onerous. In a few cases, we even resorted to manually introducing a temporary `StateT` layer exclusively for a single loop, in which case we can use the simpler combinator

```
List.forM : List α → (α → m Unit) → m Unit
```

instead:

```
do ...
   let (x, y) ← StateT.run' (x, y) $ do
     zs.forM fun z => do
       let (x, y) ← get
       let x' ← StateT.lift (f x z)
       if p x' then
         set (x', g y z)
       else
         set (x', y)
     get
   ...
```

With local mutation in hand, we would like to remove the need for such boilerplate code by adding a primitive syntax for iteration to **do** blocks with full support for mutable variables.

```
do let mut x := ...
   let mut y := ...
   ...
   for z in zs do
     x ← f x z
     if p x then
       y := g y z
   ...
```

We remark the second **do** keyword in the example above is not the beginning of a nested **do** block, but part of the **for** statement syntax. Naturally, we would like to extend support for **return** to **for** as well, providing us with a succinct way to reimplement some well-known combinators.

```
def List.findM? (p : α → Bool) (xs : List α) : m (Option α) := do
  for x in xs do
    if p x then
      return some x
  return none
```

Depending on the context and whether partial application can be used, it might even be more natural to inline these small definitions (adjusting the use of **return** if necessary) instead of remembering the combinator's name and calling it. In the supplement, we prove by induction and simplification that these implementations are equivalent to the recursive definitions. Finally, in line with **return** we would also like to support **break** and **continue** with the expected semantics.

We give the formal syntax and translation supporting all these features in Figure 8. The main translation rule (D5) introduces two exception effects using `runCatch`: one for **break** outside, and one for **continue** inside a call to a collection-polymorphic `forM`, meaning both the loop body and the overall loop return `Unit` in their respective monad. The loop body s is then rewritten appropriately, starting with the outer effect (so that its actions will be lifted by the subsequent rewrite for the inner effect) using the helper function B. B rewrites any **break** statement to `throw ()` (B1) and lifts all other monadic actions (B3), much like R. However, it should not rewrite **break** in nested loops (B8), so at that point we switch to another helper function L that merely lifts the loop into the correct monad. The helper function C for the inner effect is defined analogously to B.

Finally, we adapt the existing extensions to the newly introduced syntax: for R, this is merely recursive traversal (R7-R9), but for S_y, we need to introduce a `get` call at the loop entrance since it is a control flow join point (S9).

Before continuing, let us ensure that the order of effects indeed make sense: in the most general case, a **for** loop can contain all of **return**, reassignments to outer mutable variables, **break**, **continue**, and inner mutable variables, which correspond to effects introduced on top of the base monad in this order. While state effects and exception effects commute with instances of the same kind, this is not true when combining the two different kinds: an exception monad transformer on top of a state transformer will preserve the current state on an exception, whereas it will be lost when stacked in reverse. Thus the state of inner mutable variables, but not outer ones, will be lost on **continue** or **break**, while all state will be lost on **return**, which matches our intuitive understanding of these imperative concepts.

Figure 9 contains the translation of a small **do** block as an example. We can see that the top-most exception effect is used in place of **continue**, the one below for **break**, and that all other actions are lifted to the base monad below both of them.

We note that while B and C could readily be fused into a single pass, separating them enables us to conditionally execute only one of the two passes (or neither of them) depending on the presence of the respective command in the loop. If a pass is not needed, the respective `runCatch` call should be removed as well. Finally, the `get` call can also be elided if there is no reassignment of the mutable variable in question inside the loop body.

In our encoding, the collection-polymorphic `forM` is parametrized by the type class `ForM` γ α, where γ is some container type of elements of type α.

```
forM [Monad m] [ForM γ α] : γ → (α → m Unit) → m Unit
```

We remark that, in Lean, square brackets indicate that the arguments of type `Monad m` and `ForM` γ α are *instance-implicit*, i.e. that they should be synthesized using typeclass resolution [Wadler and Blott 1989]. A function with a similar signature exists in the `Foldable` t typeclass of the Haskell base library, where $\gamma = t\ \alpha$. Alternatively, the `MonoFoldable` typeclass of the `mono-traversible` package[5]

[5] https://hackage.haskell.org/package/mono-traversable-1.0.15.1/docs/Data-MonoTraversable.html

Syntax

$$
\begin{aligned}
s \in \mathit{Stmt} ::= {} & \ldots \\
& \mid \mathbf{break} \\
& \mid \mathbf{continue} \\
& \mid \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s
\end{aligned}
$$

Translation

$$D(\mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s) = \mathtt{runCatch}\ (\mathtt{forM}\ e\ (\mathbf{fun}\ x \Rightarrow \mathtt{runCatch}\ D(C(B(s))))) \tag{D5}$$

$B : \mathit{Stmt} \longrightarrow \mathit{Stmt}$

$$
\begin{aligned}
B(\mathbf{break}) &= \mathtt{throw}\ () && \text{(B1)}\\
B(\mathbf{continue}) &= \mathbf{continue} && \text{(B2)}\\
B(e) &= \mathtt{ExceptT.lift}\ e && \text{(B3)}\\
B(\mathbf{let}\ x \leftarrow s;\ s') &= \mathbf{let}\ x \leftarrow B(s);\ B(s') && \text{(B4)}\\
B(\mathbf{let}\ \mathbf{mut}\ x := e;\ s) &= \mathbf{let}\ \mathbf{mut}\ x := e;\ B(s) && \text{(B5)}\\
B(x := e) &= x := e && \text{(B6)}\\
B(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) &= \mathbf{if}\ e\ \mathbf{then}\ B(s_1)\ \mathbf{else}\ B(s_2) && \text{(B7)}\\
B(\mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s) &= \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ L(s) && \text{(B8)}
\end{aligned}
$$

$L : \mathit{Stmt} \longrightarrow \mathit{Stmt}$

$$
\begin{aligned}
L(\mathbf{break}) &= \mathbf{break} && \text{(L1)}\\
L(\mathbf{continue}) &= \mathbf{continue} && \text{(L2)}\\
L(e) &= \mathtt{ExceptT.lift}\ e && \text{(L3)}\\
L(\mathbf{let}\ x \leftarrow s;\ s') &= \mathbf{let}\ x \leftarrow L(s);\ L(s') && \text{(L4)}\\
L(\mathbf{let}\ \mathbf{mut}\ x := e;\ s) &= \mathbf{let}\ \mathbf{mut}\ x := e;\ L(s) && \text{(L5)}\\
L(x := e) &= x := e && \text{(L6)}\\
L(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) &= \mathbf{if}\ e\ \mathbf{then}\ L(s_1)\ \mathbf{else}\ L(s_2) && \text{(L7)}\\
L(\mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s) &= \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ L(s) && \text{(L8)}
\end{aligned}
$$

$$
\begin{aligned}
S_y(\mathbf{break}) &= \mathbf{break} && \text{(S7)}\\
S_y(\mathbf{continue}) &= \mathbf{continue} && \text{(S8)}\\
S_y(\mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s) &= \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ (\mathbf{let}\ y \leftarrow \mathtt{get};\ S_y(s)) && \text{(S9)}
\end{aligned}
$$

$$
\begin{aligned}
R(\mathbf{break}) &= \mathbf{break} && \text{(R7)}\\
R(\mathbf{continue}) &= \mathbf{continue} && \text{(R8)}\\
R(\mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s) &= \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ R(s) && \text{(R9)}
\end{aligned}
$$

Fig. 8. A **do** with support for iteration

```
do let mut s := 0
   for x in xs do
     if x % 2 = 0 then
       continue
     if x > 5 then
       break
     s := s + x
   IO.println s
```

```
runCatch
  (let s := 0 in
   StateT.run' (do
       runCatch (forM xs (fun x => runCatch (do
         let s ← ExceptT.lift (ExceptT.lift get)
         if x % 2 = 0 then throw ()
         else ExceptT.lift (ExceptT.lift (StateT.lift (ExceptT.lift (pure ()))))
         let s ← ExceptT.lift (ExceptT.lift get)
         if x > 5 then ExceptT.lift (throw ())
         else ExceptT.lift (ExceptT.lift (StateT.lift (ExceptT.lift (pure ()))))
         let s ← ExceptT.lift (ExceptT.lift get)
         ExceptT.lift (ExceptT.lift (set (s + x))))))
       let s ← get
       StateT.lift (ExceptT.lift (IO.println s)))
     s)
```

Fig. 9. Translation example, using basic **do** notation instead of >>= for readability

provides a function that is closer to the above signature and as such also allows for iterating over monomorphic collection types as well as ones with more than one type parameter, such as the key-value pairs of a map, without going through a temporary list.

Compared to other looping constructs, using fold-like traversal as a primitive is particularly interesting for total languages such as Lean because it delegates the issue of termination to the combinator. Indeed, if we opt into Lean's support for non-total functions, we can introduce the **repeat** and **while** statements as two syntax abbreviations

```
repeat s       ≡  for u in Loop.mk do s
while c do s   ≡  repeat ((unless c do break); s)
```

where Loop is an auxiliary type containing a single constructor Loop.mk : Loop, and its ForM Loop Unit instance is defined using the function

```
  partial def loopForever [Monad m] (f : Unit → m Unit) : m Unit :=
    f () >>= fun _ => loopForever f
```

The **partial** keyword ensures soundness by checking that the function type is inhabited (in this case by **fun** f => pure ()) and then turning the function into an opaque constant, making its body inaccessible to proofs.

5 IMPLEMENTATION

5.1 Reference Implementation

We have implemented our extended `do` notation in Lean 4. The supplemental material contains a reference implementation, examples, and equivalence proofs. Our reference implementation relies on Lean's hygienic macro system [Ullrich and de Moura 2020], and is a direct encoding of the translation rules presented in the previous sections. Languages with similarly expressive macro systems should allow for easy adaptation of the implementation.

We start by defining the *syntactic category* of statements:

```
declare_syntax_cat stmt
```

Syntactic categories start out empty and can be extended with new syntax variants at any point.

In the supplemental material, we use the keyword `do'` instead of `do` to distinguish the reference implementation from Lean's full implementation. We define the `do'` parser using the following command:

```
syntax "do'" stmt : term
```

This parser specifies that the keyword `do'` followed by `stmt` is a `term`, where `term` is a builtin syntactic category. We encode all abbreviations as simple macros. For example, we implement abbreviation A2 using the macro command

```
macro:0 s1:stmt ";" s2:stmt : stmt => `(let x ← $s1; $s2)
```

The left-hand side defines a new parser for the category `stmt` with explicit precedence 0, and the right-hand side uses an explicit syntax quasiquotation to construct the syntax tree, with syntax placeholders (antiquotations) prefixed with `$`. Because the second `stmt` is not restricted to a precedence level, Lean will greedily associate the notation to the right as expected. By restricting nested statements in trailing positions to precedence levels greater than 0, we can force them not to contain a semicolon unless wrapped in curly braces, which use the implicit maximum precedence.

```
syntax "if" term "then" stmt "else" stmt:1 : stmt
macro "{" s:stmt "}" : stmt => `($s)
```

We represent the function D using the following auxiliary notation

```
syntax "d!" stmt : term      -- `d! s` corresponds to `D(s)`
```

and encode rule D2 as follows:

```
macro_rules | `(d! let $x ← $s; $s') => `((d! $s) >>= fun $x => (d! $s'))
```

Because `macro_rules` declarations may extend existing macros with new cases at any point, we can in fact introduce our language extensions modularly just like in the formal translation, with each file in the supplement corresponding to one of the extensions.

Similarly to the above rule, we encode function S_y using the auxiliary notations

```
declare_syntax_cat expander
syntax "expand!" expander "in" stmt : stmt
syntax "mut" ident : expander
```

and encode rule S1 using the following command

```
macro_rules | `(stmt| expand! mut $y in $e:term) => `(stmt| StateT.lift $e)
```

where the `stmt|` prefix adjusts the syntactic category parsed by the quasiquotations. We use the abstract *expander* syntax category above to factor out common traversing rules at functions S_y, R, B, and L. We have the following syntax declarations for the latter three functions:

```
syntax "return" : expander   syntax "break" : expander   syntax "lift" : expander
```

Then, we encode rules S6, R6, B7, and L7 described earlier using a single rule

```
macro_rules
  | `(stmt| expand! $exp in if $e then $s₁ else $s₂) =>
    `(stmt| if $e then expand! $exp in $s₁ else expand! $exp in $s₂)
```

We also avoid adding unnecessary monadic layers with a simple check: if the number of nested occurrences of the keyword in question (**return**, **break**, or **continue**) has not changed after applying the respective helper function, we throw away the transformation result and do not emit the respective runCatch code. The full implementation uses a **do**-specific abstract syntax tree that makes such analyses more efficient.

Our reference implementation produces error messages when side conditions do not hold. For example, we implement rule (S2) as follows

```
macro_rules
  | `(stmt| expand! mut $y in let $x ← $s; $s') =>
    if x == y then
      throw $ Macro.Exception.error x s!"cannot shadow 'mut' variable '{x.getId}'"
    else
      `(stmt| let $x ← expand! mut $y in $s; let $y ← get; expand! mut $y in $s')
```

5.2 Code Generation

As we can observe in Figure 9, the output produced by our translation can be quite verbose. In our first implementation, we used the standard StateT and ExceptT monad transformers. With these transformers, the Lean compiler can eliminate most of the overhead introduced by the translation by inlining the monadic operators and applying program transformations such as match-of-match and match-of-constructor [Peyton Jones 1996]. The only exception is the **for** statement. Even after specializing a forM instance, the specialized function would still allocate transient pairs for storing the updated mutable variables, and Except objects. When a particular forM instance is tail recursive, we believe GHC can eliminate this overhead completely by inlining the specialized function as a recursive join point and then performing fusion [Maurer et al. 2017]. Our current compiler does not support recursive join points, and even if it did, it would not be sufficient for non-tail recursive forM instances (e.g., for trees). We addressed this issue by using CPS versions of StateT and ExceptT monad transformers.

```
def StateCpsT (σ : Type u) (m : Type u → Type v) (α : Type u) :=
  (δ : Type u) → σ → (α → σ → m δ) → m δ
instance : Monad (StateCpsT σ m) where
  pure a   := fun δ s k => k a s
  bind x f := fun δ s k => x δ s (fun a s => f a δ s k)

def ExceptCpsT (ε : Type u) (m : Type u → Type v) (α : Type u) :=
  (β : Type u) → (α → m β) → (ε → m β) → m β
instance : Monad (ExceptCpsT ε m) where
  pure a   := fun β k₁ k₂ => k₁ a
  bind x f := fun β k₁ k₂ => x β (fun a => f a β k₁ k₂) k₂
```

and we define runCatch for ExceptCpsT as follows

```
def runCatch [Monad m] (x : ExceptCpsT α m α) : m α :=
  x α pure pure
```

Then, after simplification and code specialization, the Lean compiler produces the following recursive function for the example in Figure 9.

```
def rec xs s :=
  match xs with
  | []    => IO.println s
  | x::xs =>
    match x % 2 = 0 with
    | false =>
      match x > 5 with
      | false => rec xs (x + s)
      | true  => IO.println s
    | true  => rec xs s
```

5.3 Full Implementation

We have "optimized" our reference implementation for conciseness and simplicity, and it diverges from the full implementation built into Lean[6] in some details. First, it has no support for patterns. For example, we cannot write `do' let (a, b) ← s; ...`. The full implementation also supports **match** statements where the right-hand side of each alternative is a sequence of statements. For example, we can write

```
do let mut n := 0
   for x in xs do
     match x with
     | none => n := n + 1
     | _    => pure ()
   IO.println n
```

There is no fundamental challenge here, and the implementation of **match** statements is analogous to the one for **if** statements. Indeed, the full implementation also supports the Rust-inspired abbreviation `if let none := x then n := n + 1` of the above **match** block mixing the two styles.

We also short-circuit translation steps to speed up compilation time of the produced terms. For example, in the reference implementation, `let mut x ← s; ...` is first expanded into `let y ← s; let mut x := y; ...`. Although the compiler can simplify the code, we observed that this does increase compilation time. In the full implementation, `let mut x ← s` is not an abbreviation.

The sharp-eyed reader may have noticed that we can reassign variables in the reference implementation with values of a different type. For example, `do' let mut x := 0; x := true; ...` is accepted, but, in the full implementation, we generate the following error message

```
error: invalid reassignment, value has type
  Bool
but is expected to have type
  Nat
```

We implement this check using the Lean built-in term `ensureTypeOf! s msg e`. This term instructs the elaborator to produce the error message `msg` when the type of `s` and `e` do not match.

Our full implementation also supports two useful features that are not directly inspired by imperative languages features, but nevertheless can help in avoiding boilerplate in monadic programs absent from equivalent imperative ones, so we mention them here for the sake of completeness: nested actions and automatic monadic lifting. The first feature allows users to embed terms of the form `← a` in expressions, and is similar to the `!a` notation available in the Idris programming language [Brady 2013]. For example,

```
do if (← get).x >= 0 then
     action
```

[6] https://github.com/leanprover/lean4/blob/v4.0.0-m4/src/Lean/Elab/Do.lean

expands to

```
do let s ← get
   if s.x >= 0 then
     action
```

The main difference to Idris' implementation is that we wanted to make the scope of nested actions more predictable: instead of being lifted "as high as possible" [Brady 2014], they are always lifted to the enclosing **do** block. Using the notation outside of a **do** block is an error.

The second feature, automatic monadic lifting, extends the implicit coercion insertion procedure used in Lean. Most theorem provers support implicit coercions which map elements of one type to another. For example, a coercion from `Nat` to `Int` allows a natural number value to be used where an integer is expected. The implicit coercions are automatically inserted by the elaborator to address type mismatches. In Lean, we have the typeclass

```
class MonadLift (m : Type → Type) (n : Type → Type) where
  monadLift : m α → n α
```

which is based on the homonymous typeclass available in the `Control.Monad.Layer` Haskell package. The `monadLift` operation can be viewed as a coercion. For example, assume we have the method `inferType : Expr → MetaM Expr` to retrieve the type of a given expression, we are writing a function in the monad `ElabM`, and we have an instance `MonadLift MetaM ElabM`. Then, we used to write `monadLift (inferType e)`, but we found it to be quite tedious to add these lifting operations manually. Automatic monadic lifting automatically inserts the `monadLift` operations when needed.

6 REASONING

One of the stated advantages of purely functional programming is ease of reasoning. Fortunately, Lean is also a theorem prover (based on the Calculus of Inductive Constructions [Coquand and Huet 1988; Coquand and Paulin 1990]) and thus can be used to show that the output of our translation, while (sometimes unnecessarily) verbose, can still be analyzed using the same tools as corresponding code not using the extensions and indeed shown to be equivalent to it. We have included such equivalence proofs in the supplement material. Our proofs hold for any monad `m` that has an instance `LawfulMonad m`. The typeclass `LawfulMonad` encapsulates the monadic laws of Figure 2. These equivalence proofs are straightforward using Lean's tactic framework for constructing proofs. Tactics are user-defined or built-in Lean functions that construct terms representing proofs. The main tactics used in our examples are `cases` (for case-analysis), `induction`, and `simp`. The latter is a term simplifier that uses previously proved theorems marked with the attribute `@[simp]` as simplification rules. The monadic (Figure 2), `StateT` (Figure 4), and `ExceptT` (Figure 6) laws are marked with the attribute `@[simp]`. The Lean standard library also contains additional simplification lemmas that follow from these laws. For example, it contains

```
@[simp] theorem bind_pure_unit [Monad m] [LawfulMonad m] (x : m Unit)
        : (x >>= fun _ => pure ()) = x := ...
```

The simplifier `simp` is crucial for automating all proofs in the supplemental material because of the mentioned verbosity of the output produced by our translation rules. In the following example, we show that a monadic program constructed using our **do**' notation is equal to

```
ma >>= fun x => if b then ma' else pure x
```

```
theorem simple [Monad m] [LawfulMonad m] (b : Bool) (ma ma' : m α) :
    (do' let mut x ← ma;
         if b then { x ← ma' };
         pure x)
    = (ma >>= fun x => if b then ma' else pure x) :=
  by cases b <;> simp
```

The tactic cases b splits the proof into two cases (b = true) and (b = false), and both of them are then solved using simp. We use induction to prove properties of programs containing the **for** iterator. In the following theorem, we prove that a program with two nested **for** statements is equal to List.findSomeM? (**fun** xs => List.findM? p xs) xss, where List.findSomeM? and List.findM? are defined as recursive functions.

```
theorem eq_findSomeM_findM [Monad m] [LawfulMonad m] (xss : List (List α)) :
      (do' for xs in xss do' {
             for x in xs do' {
               let b ← p x;
               if b then { return some x }
           } };
           pure none)
      = List.findSomeM? (fun xs => List.findM? p xs) xss := by
  induction xss with
  | nil => simp [List.findSomeM?]
  | cons xs xss' ih =>
    simp [List.findSomeM?]
    rw [← ih, ← eq_findM]
    induction xs with
    | nil => simp
    | cons x xs' ih => simp; apply byCases_Bool_bind <;> simp [ih]
```

The induction tactic induction xss **with** ... splits the proof into two cases (xss = []) and (xss = xs::xss'), and the variable ih corresponds to the induction hypothesis in the latter. The tactic simp [List.findSomeM?] not only applies theorems marked with the @[simp] attribute, but also unfolds the definition List.findSomeM?. The rewrite tactic rw applies a sequence of theorems as rewriting rules. The modifier argument ← instructs Lean to apply symmetry before rewriting. For example, the theorem eq_findM, also included in the supplemental material, is a proof that

```
do' for x in xs do' {
      let b ← p x;
      if b then { return some x }
    };
    pure none
```

is equal to xs.findM? p. Thus, the parameter ← eq_findM instructs Lean to replace an occurrence of xs.findM? p in the proof goal with the **do**' block above. We use the tactic apply byCases_Bool_bind to perform case analysis on the result of an action that produces a Boolean value, where the theorem byCases_Bool_bind has type

```
theorem byCases_Bool_bind
  [Monad m] [LawfulMonad m] (x : m Bool) (f g : Bool → m β)
  (isTrue : f true = g true) (isFalse : f false = g false) : (x >>= f) = (x >>= g)
```

That is, the monadic programs x >>= f and x >>= g are equal if f true = g true and f false = g false. Finally, we close the two branches by using simp with the inner induction hypothesis as an additional simplification rule.

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

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

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

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

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

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

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

$$\frac{e[\sigma] \Rightarrow \texttt{true} \qquad \langle s_1, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle} \quad \frac{e[\sigma] \Rightarrow \texttt{false} \qquad \langle s_2, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}$$

$$\frac{\begin{array}{c} e[\sigma] \Rightarrow \texttt{cons}\ v_1\ v_2 \\ \langle s[v_1/x], \sigma \rangle \Rightarrow \langle n, \sigma' \rangle \qquad n \in \{\texttt{()}, \mathbf{continue}\} \qquad \langle \mathbf{for}\ x\ \mathbf{in}\ v_2\ \mathbf{do}\ s, \sigma' \rangle \Rightarrow \langle n, \sigma'' \rangle \end{array}}{\langle \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s, \sigma \rangle \Rightarrow \langle n, \sigma'' \rangle}$$

$$\frac{e[\sigma] \Rightarrow \texttt{cons}\ v_1\ v_2 \qquad \langle s[v_1/x], \sigma \rangle \Rightarrow \langle \mathbf{break}, \sigma' \rangle}{\langle \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s, \sigma \rangle \Rightarrow \langle \texttt{()}, \sigma' \rangle}$$

$$\frac{e[\sigma] \Rightarrow \texttt{cons}\ v_1\ v_2 \qquad \langle s[v_1/x], \sigma \rangle \Rightarrow \langle \mathbf{return}\ v, \sigma' \rangle}{\langle \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ s, \sigma \rangle \Rightarrow \langle \mathbf{return}\ v, \sigma' \rangle}$$

Fig. 10. Extending a natural semantics $e \Rightarrow v$ reducing expressions to values with a rule for **do** block evaluation for the special case of the identity monad. A helper relation $\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle$ reduces statements to *neutral* statements under a mutable state context (a partial map from variables to values, updated via the notation $\cdot[\cdot \mapsto \cdot]$). Immutable bindings are evaluated by immediate substitution in the remainder statement, while the mutable context is substituted ($e[\cdot]$) just before evaluating any nested term e. The given value type and semantics are *strict* (see cons), but could easily be changed to be lazy.

7 FORMALIZATION

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

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

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

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

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

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

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

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

Fig. 11. Extending an expression typing relation $\Gamma \vdash e : \tau$ with a rule for typing **do** blocks via a statement typing relation $\Gamma \mid \Delta \vdash_f s : m\ \alpha \hookrightarrow \omega$ over some monad m. Δ is an additional context of mutable variables, initially empty. ω is the *return type* expected inside **return** statements, initially equal to α but may diverge from it in **let** bindings. $f \in \{\mathtt{for}, \mathtt{nofor}\}$ controls occurrences of **break** and **continue**.

```
inductive Stmt (m : Type → Type) (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
  | letmut (e : Γ ⊢ Δ ⊢ α) (s : Stmt m ω Γ (α :: Δ) b c β) : Stmt m ω Γ Δ b c β
  | assg (x : Fin Δ.length) (e : Γ ⊢ Δ ⊢ Δ.get x) : Stmt m ω Γ Δ b c Unit
  | ite (e : Γ ⊢ Δ ⊢ Bool) (s₁ s₂ : Stmt m ω Γ Δ b c α) : Stmt m ω Γ Δ b c α
  | ret (e : Γ ⊢ Δ ⊢ ω) : Stmt m ω Γ Δ b c α
  | sfor (e : Γ ⊢ Δ ⊢ List α) (s : Stmt m ω (α :: Γ) Δ true true Unit) :
      Stmt m ω Γ Δ b c Unit
  | sbreak : Stmt m ω Γ Δ true c α
  | scont : Stmt m ω Γ Δ b true α
```

Fig. 12. Inductive family representing intrinsically typed **do** statements with shallowly embedded terms. The notation $\Gamma \vdash \Delta \vdash \alpha$ stands for the type of functions mapping values (two heterogeneous lists) corresponding to the types in Γ and Δ to a value of α. Each constructor implements and corresponds to, in order, a typing rule from Figure 11. Of particular note are constructors changing indices of the type family: `bind` and `sfor` extend the immutable context while `letmut` extends the mutable context, which is accessed by `assg` (assignment), representing the target variable as an index into the context, i.e. a de Bruijn index.

While the previous section demonstrated that specific examples of the extended **do** notation can be shown to correspond to their expected semantics, that does not conclusively prove that our translation produces sensible — or even type-correct — output in all cases. Most of our translation rules are relatively straightforward, but there are subtle details around variable binding and shadowing as well as layering and lifting of monad transformers so that we cannot a priori exclude the possibility of mistakes in them.

In order to increase our trust in the translation's correctness, we have formulated an operational semantics of our **do** notation that gives an alternative, dynamic and even simpler view of the expected behavior (Figure 10) as well as a corresponding type system that formalizes the expected static semantics of the notation (Figure 11) such that we expect the following correctness theorem to hold.

Theorem. *For any well-typed* **do** *block* $\Gamma \vdash$ **do** `s` $: \tau$*, there exists a unique (up to* α*-equivalence) translation* **do** `s` $\rightsquigarrow e$*, which is of the same type (*$\Gamma \vdash e : \tau$*) and equivalent under evaluation (*$\forall v.$ **do** `s` $\Rightarrow v \longleftrightarrow e \Rightarrow v$*).*

Proof. See below for a strictly stronger statement for the case of Lean as the base language of terms, formalized and proved in Lean.

We necessarily restrict the semantics to the special case of the identity monad and iteration over lists (arbitrarily presented as strict, like in the Lean language), as we would not be able to formulate the rules as such in presence of arbitrary monad and `ForM` instances. Since the translation functions are generic over these instances, we believe this is only a minor restriction. The semantics and type system are otherwise generic over those of the base language, and thus a proof of the correctness theorem is also dependent on the base semantics, including an implementation of all monad transformer functions that fulfills the previously presented equations.

In order to focus on the translation and semantics of the **do** notation instead of these details of the base language, we decided to verify the correctness theorem in Lean using the Lean language itself and its existing implementation of monad transformers as the representation of terms, that is, a *shallow embedding* of Lean terms (abstracted over Γ and Δ) inside of a *deep embedding* of **do** statements as an inductive datatype. More specifically, statements are represented *intrinsically typed* by an inductive family `Stmt m` ω Γ Δ `b c` α such that the presented typing rules are fulfilled by definition (Figure 12). The parameters of the type constructor correspond to the variables of the same name in Figure 11, except that we split f into `b, c : Bool` that separately control **break** and **continue**, respectively, for reasons that will become clear in the next paragraph. As is common with formalizations, we additionally do not use named variables, but choose de Bruijn notation so that Γ and Δ are lists of types and variable references indices into those lists, trivializing α-equivalence. Bindings outside of the **do** block are not part of Γ but represented as regular Lean bindings so that Γ is initially empty. Thus the statement

```
let x ← ...; let mut y := ...; y := pure (x + y)
```

is represented by the Lean term

```
Stmt.bind ... (Stmt.letmut ... (Stmt.assg 0 (fun ([x]) ([y]) => pure (x + y))))
```

where the context assignments are destructed into the individual variables by pattern matching on the heterogeneous lists. The full definitions and proofs available in the supplement [Ullrich and de Moura 2022] are written in a literate style explaining further details and exported to interactive HTML via *Alectryon* [Pit-Claudel 2020]/*LeanInk* [Bülow 2022], giving access to type and goal information without having to install Lean.

This very precise representation of statements not only guarantees that the translation functions always produce unique, type-correct statements and terms as long as they are themselves type-correct Lean functions, but it also tells us much, and gives guarantees about, their working just by looking at their signatures:

```
S [Monad m] : Stmt m ω Γ (Δ ++ [α]) b c β → Stmt (StateT α m) ω (α :: Γ) Δ b c β
R [Monad m] : Stmt m ω Γ Δ b c α → Stmt (ExceptT ω m) Empty Γ Δ b c α
L [Monad m] : Stmt m ω Γ Δ b c α → Stmt (ExceptT PUnit m) ω Γ Δ b c α
B [Monad m] : Stmt m ω Γ Δ b c α → Stmt (ExceptT PUnit m) ω Γ Δ false c α
C [Monad m] : Stmt m ω Γ Δ false c α → Stmt (ExceptT PUnit m) ω Γ Δ false false α
D [Monad m] : Stmt m Empty Γ ∅ false false α → (Γ ⊢ m α)
```

We can immediately see that `S` transforms a mutable variable (always the outermost one, so there is no need for the parameter y with de Bruijn notation) to an immutable variable, that `R` eliminates **`return`** statements (by setting their expected type to the uninhabited type `Empty`), and B and C **`break`** and **`continue`** statements, respectively, as well what monadic layers they each introduce. Finally, `D` transforms a statement free of **`return`** and unbounded occurrences of mutable variables, **`break`**, and **`continue`** to a term of the expected type.

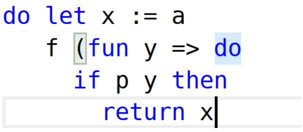
The choice of shallowly embedded terms obviates the dependency on a presentation of the natural semantics for the base language, and analogously we can express evaluation of statements directly as a denotational function ⟦·⟧ `: Do Id α → α` (where `Do m α` is an abbreviation of `Stmt m α ∅ ∅ false false α`) instead of an inductive predicate. Lean again guarantees that evaluation of any well-typed statement is unambiguous and terminating this way, which are desirable properties that we would expect to hold as long as the base language also fulfills them. In fact, the implementation as a function makes it trivial to lift the restriction on the base monad and strengthen ⟦·⟧ into the type `[Monad m] → Do m α → m α`. This generalization is in fact crucial for our proof of the correctness theorem because it allows us to verify each translation function individually and modularly, finally composing the correctness proofs of `R` and `D` into the following succinct generalization of the correctness theorem over any monad obeying the laws from Figure 2.

```
def Do.trans [Monad m] (s : Do m α) : m α := runCatch (D (R s) ∅)  -- (1′)
theorem Do.trans_eq_eval [Monad m] [LawfulMonad m] :
  ∀ s : Do m α, Do.trans s = ⟦s⟧
```

Proving its correctness is not the only application of `Do.trans`, however. While our choice of mixed deep/shallow embedding by design blurs the distinction between the translated language and the implementation language, we can still see two separate *stages* in the signature of the main translation function, `D`: it first takes a `Stmt`, i.e. compile-time information, and then an assignment of the immutable context Γ, that is, run-time information. Indeed, it is possible to *partially* evaluate `D` and the other translation functions given a `Stmt` but not the context assignment (or values of any other variables free in expressions nested in `Stmt`) such that the `Stmt` is completely erased. While Lean does not natively support this kind of multi-stage programming, we demonstrate in the supplement how the built-in simplifier can be used as a partial evaluator for this purpose. Thus the formal translation could be used to replace the macro implementation while providing more static guarantees, though the latter is still more desirable in terms of efficiency and modularity.

8 EVALUATION

In this section, we describe our experience with the extended **do** notation. We are developing Lean 4 in Lean itself. In October of 2020, we finally compiled Lean 4 using itself and started using the new notation in our codebase. We believe it is making us more productive, and have already refactored many existing **do** blocks into ones making use of the new features. The user feedback so far has



```
do let x := a
   f (fun y => do
     if p y then
       return x
```

Fig. 13. Highlighting the **do** keyword belonging to a **return** statement under the cursor using the standard "document highlight" request of the Language Server Protocol[8], shown here using Visual Studio Code

also been very positive, and the extended **do** notation is used in many applications and packages developed by the Lean community: out of 43 GitHub repositories written in Lean with the topic "lean4"[7], 31 repositories by 20 different authors make use of at least **let mut** and **for**. Thus we feel safe to claim that the use of extended **do** notation in Lean 4 programming has by now become ubiquitous.

At the time of writing, the Lean 4 codebase itself contains 459 occurrences of **let mut** declarations, and 604 occurrences of **for**. The codebase also contains 3185 occurrences of **return**, though many of them are equivalent to `pure`, i.e. they do not actually short-circuit evaluation but merely avoid the need for parentheses around the return value.

In our first implementation of the extended **do** notation, we did not have the **mut** modifier, and any variable could be locally mutated. We assumed it would make the system more convenient to use. However, we have reverted this design decision after we encountered a few non-trivial bugs in our codebase. All bugs occurred in code containing nested **do** blocks such as

```
do let x := a
   ...
   f (fun y => do
      ...
      x := b
      ...)
   ...
```

In the example above, the user intended to update the variable `x` in the outer **do** block, but the reassignment `x := b` is in the *scope* of a different **do**. We say the mistake was due to "scope confusion." Our new procedure, described earlier in this paper, prevents this kind of mistake because it produces an error message if the nested block does not contain a `let mut x :=` ... declaration. We also remark the **mut** keyword dramatically simplifies the implementation by making scope checks a simple decision local to **do** notation as manifested in the helper function S_y. We conjecture scope confusion may also occur when using **return**. In practice, this does not seem to be an issue since the expected type for the nested **do** block is often different from the outer one. However, as a precaution, we have implemented highlighting of the corresponding **do** keyword when users place their cursor on a **return** statement in any editor supporting the Lean 4 language server (Figure 13).

We are pleasantly surprised that users are also using the extended **do** notation in pure code via the identity monad. For example, the `wrapAt?` function in the `Cli` package[9] uses the following **for** statement.

[7] https://github.com/topics/lean4
[8] https://microsoft.github.io/language-server-protocol
[9] https://github.com/mhuisi/lean4-cli

```
Id.run $ do ...
   let mut line := line
   let mut result := #[]
   for i in [:resultLineCount] do
     result := result.push (line.take maxWidth)
     line := line.drop maxWidth
   return "\n".intercalate result.toList
```

The notation `#[]` denotes the empty array, and `[:resultLineCount]` is a range of natural numbers from `0` up to `resultLineCount` (exclusively).

Users also seem to prefer the **for** statement even when it corresponds to an existing combinator such as `foldl`. For example, the translation verifier `reopt-vcg`[10] contains the following code fragment.

```
do ...
   let mut stats : GoalStats := GoalStats.init
   for r in results do
     stats := stats.addResult r
   pure stats
```

As a final example, we give the function `hasBadParamDep?` from the Lean 4 codebase combining nested iteration, nested actions, and early return.

```
def hasBadParamDep? (ys : Array Expr) (indParams : Array Expr)
    : MetaM (Option (Expr × Expr)) := do
  for p in indParams do
    let pType ← inferType p
    for y in ys do
      if (← dependsOn pType y) then
        return some (p, y)
  return none
```

We can use the same approach we used in theorem `eq_findSome_findM` to prove that it is equivalent to a function defined using `findSomeM?` and `findM?`.

9 RELATED WORK

We are not aware of similar formal work exploring imperative extensions to a purely functional language, but there are multiple existing libraries with some overlap. Perhaps the library closest to our work is the proof-of-concept Haskell package *ImperativeHaskell*[11], which via creative use of custom operators encodes mutable variables, **for** counting loops, and early return. The implementation is based on mutable references (`IORef`) and limited to `IO` as the base monad, which is a significant restriction in practice that in particular would severely complicate verification of programs using it. We believe that any implementation of these features on top of arbitrary monads would require more expressive desugaring than custom operators such as presented in this paper. The *early* library[12] makes use of a GHC plugin to provide a syntax for early return when binding particular values, inspired by a similar syntax in Rust. The *control-monad-loop* library[13] provides a looping function with **continue** and **break** functionality encoded via a continuations-carrying monad, but none of the other effects. It also supports returning values from **break**s, which we have considered but discarded for the time being for lack of convincing use cases.

[10] https://github.com/GaloisInc/reopt-vcg

[11] https://hackage.haskell.org/package/ImperativeHaskell

[12] https://github.com/inflex-io/early

[13] https://hackage.haskell.org/package/control-monad-loop-0.1

Apart from these imperative extensions, we are aware of three further extensions of Haskell's **do** notation: Marlow et al. [2016] optimize its desugaring so that some blocks can be run using only `Applicative` instead of `Monad` operations, making **do** blocks both more general and potentially more efficient. Erkök and Launchbury [2002] add an **mdo** variant of the notation that changes the semantics of monadic bindings to allow recursion. Both extensions have since been implemented as language extensions of the GHC compiler, and should be compatible with our imperative extensions. Paterson [2001] adapts the notation to *arrows*, a generalization of monads, which we have not explored in Lean so far.

Outside of Haskell, the Scala library *effectful*[14] translates **for** loops in a **do**-like macro to applications of `traverse`, but does not combine this with support for further control flow like **break**, **continue**, and **return**, or for local mutation. Idris features an extended **do** notation [Brady 2014] that allows giving "alternative" patterns for a binding, which if matched determine the result of the whole block without executing the remaining statements:

```
do Just x_ok ← readNumber | Nothing => pure Nothing
   Just y_ok ← readNumber | Nothing => pure Nothing
   pure (Just (x_ok, y_ok))
```

This can be seen as an implementation of early return, though without nesting in further control flow statements such as **if** or **for**. A similar syntax was later added to Agda [Bove et al. 2009]. The Koka language [Leijen 2014] is a function-oriented language with built-in effects and as such does not employ monads or **do** notation. However, we note that it has support for both mutable variables and multi-shot effects such as nondeterminism, for the combination of which it has assigned the same "strand-local" semantics as we discussed in Section 2.

Gibbons and dos Santos Oliveira [2009] identify *mapping* and *accumulating* as the core aspects of imperative iteration, and show that the `traverse` operator can represent both of them simultaneously in functional code. In our paper, we focused on the more restrictive folds, which embody only the accumulating part, since they cover most use cases where local mutation or extended control flow can profitably be applied in our experience. However, it is possible to extend our approach to `traverse`, e.g. with a new `for mut x in xs do s` syntax that for a mutable variable xs allows x to be reassigned in the loop body and eventually reassigns the thus mapped collection to xs.

The idea of rewriting code using mutable variables into equivalent pure code to make it amenable to formal verification has previously been explored by this paper's first author in Ullrich [2016] in the context of translating a subset of Rust to Lean. As in our translation, mutation in straight-line code is translated to shadowing, though for conditional statements, the continuation is duplicated, which is less of an issue when compilation of the translated code is not a goal. Similarly, both terminating and non-terminating loops are supported via a fold-like monadic loop combinator, but the combinator is not computable (i.e. executable) because it employs classical logic to "decide" termination. Termination must instead be proved or disproved after translation. The translation also handles some advanced cases like turning mutable references into lenses [Foster et al. 2007], which we have no plans of supporting for our use cases. Ho and Protzenko [2022] recently introduced a similar but more general approach for verification of Rust code.

Nipkow [1998] utilizes a formalization approach very similar to our mixed deep/shallow embedding, calling it "taking the semantic view". The lack of dependent types in Isabelle/HOL, however, would complicate representing a heterogeneous context like we did.

[14] https://github.com/pelotom/effectful

10 CONCLUSION

We have extended **do** notation with further features inspired by imperative programming by giving a translation to purely functional programs that can still be reasoned over, and compiled into efficient executable code. Our experience, and those of third parties, with an implementation in the Lean 4 programming language and theorem prover suggests that the extended notation is quickly embraced and leads to simple-to-understand code that can be less abstract but often no less terse than equivalent code using combinators. While implementing the presented extensions ad-hoc as demonstrated using Lean’s expressive macro system might not be possible in most other theorem provers or programming languages, we believe that a built-in implementation as described in Section 5.1 would be a worthwhile addition to other functional languages as well.

Just like McBride and Paterson [2008] state that

> “The explosion of categorical structure in functional programming [...] should not, we suggest, be a cause for alarm. Why should we not profit from whatever structure we can sniff out, abstract and re-use? The challenge is to avoid a chaotic proliferation of peculiar and incompatible notations. If we want to rationalise the notational impact of all these structures, perhaps we should try to recycle the notation we already possess.”

we say: the embrace of imperative patterns in purely functional programming where reasonable and not at odds with functional principles should not, we suggest, be a cause for alarm. Why should we not profit from whatever control flow pattern we can sniff out, abstract and re-use? The challenge is to avoid a chaotic proliferation of peculiar and incomprehensible combinators. If we want to lessen the cognitive impact of categorical structures, perhaps we should try to recycle the imperative notation programmers already possess familiarity with.

ACKNOWLEDGMENTS

We thank Sebastian Graf, Jakob von Raumer, and the anonymous reviewers for their extensive comments, corrections, and advice.

REFERENCES

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

Edwin Brady. 2014. Resource-dependent algebraic effects. In *International Symposium on Trends in Functional Programming*. Springer, 18–33.

Niklas Bülow. 2022. Proof Visualization for the Lean 4 Theorem Prover.

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Inform. and Comput.* 76, 2-3 (1988), 95–120.

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*. Springer, Berlin, 50–66.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *International Conference on Automated Deduction*. Springer, 625–635.

Levent Erkök and John Launchbury. 2002. A recursive do for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 29–37.

J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007), 17–es.

Jeremy Gibbons and Bruno César dos Santos Oliveira. 2009. The essence of the iterator pattern. *Journal of functional programming* 19, 3 and 4 (2009).

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. (2022). To appear at *International Conference on Functional Programming (ICFP)*.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Richard A Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices* 30, 3 (1995), 13–22.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (jun 2014), 100–126. https://doi.org/10.4204/eptcs.153.8

Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. *ACM SIGPLAN Notices* 51, 12 (2016), 92–104.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 482–494. https://doi.org/10.1145/3062341.3062380

Conor McBride and RA Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Tobias Nipkow. 1998. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing* 10 (1998), 171–186.

Ross Paterson. 2001. A new notation for arrows. *ACM SIGPLAN Notices* 36, 10 (2001), 229–240.

Simon L. Peyton Jones. 1996. Compiling Haskell by program transformation: a report from the trenches. In *Proc. European Symp. on Programming*. Springer-Verlag, 18–44.

Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 155–174. https://doi.org/10.1145/3426425.3426940

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification*. Master's thesis. Karlsruher Institut für Technologie (KIT).

Sebastian Ullrich and Leonardo de Moura. 2020. Beyond notations: Hygienic macro expansion for theorem proving languages. In *International Joint Conference on Automated Reasoning*. Springer, 167–182.

Sebastian Ullrich and Leonardo de Moura. 2022. *Supplement of "'do' Unchained: Embracing Local Imperativity in a Purely Functional Language"*. https://doi.org/10.5281/zenodo.6606640

Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. 61–78.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 60–76.