Don't Sweat Interaction Trees

Proof-Guided Local Variable Lifting for Interaction Trees

Yiming Lin
Portland State University, OR, USA

Ian Kariniemi
Portland State University, OR, USA

Yao Li
Portland State University, OR, USA

Abstract

Verifying existing software is hard: Programs are developed in languages not amenable to verification, involve complicated optimizations that obscure the underlying logic, and are gigantic in size. In this paper, we propose a way to ease this pain via a simplification framework that employs interaction trees as a language-agnostic interface. We show that local variable lifting, the technique underlying AutoCorres for the Simpl language, can be generalized to interaction trees via an implementation in Rocq. A key challenge with simplifying interaction trees is that they are highly dynamic structures and we would like our approach to work with mostly uninterpreted trees. We address this challenge via metaprogramming. Our metaprogramming framework is semi-automatic and proof-guided, *i.e.,* we obtain the simplified code via a constructive proof of equivalence that can be automated via proof tactics, by utilizing Rocq's `Derive` extension. This approach gives us simplified code and the equivalence theorem in one step. We demonstrate that our approach is practical using examples inspired by real-world applications.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Software verification

Keywords and phrases interaction trees, formal verification, metaprogramming

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.16

Supplementary Material *Software (Source code)*: `https://doi.org/10.5281/zenodo.20337843`

Funding This work was partially supported by the National Science Foundation under Grant No. CCF-2348490. Ian Kariniemi was partially funded by Bloomberg L.P. as a Fellow in the Bloomberg Infrastructure & Security Ph.D. Fellowship Program.

Acknowledgements We thank all anonymous reviewers of ITP'26, whose feedback has helped greatly improve this paper. We thank Mike Dodds, Mark P. Jones, Andrew Tolmach, and Fei Xie for their feedback to this work, and Alex Hodges and Allison Naaktgeboren for their feedback to this paper.

Declaration about GenAI/LLM use We developed most of the code in our supplementary materials manually. We used GitHub CoPilot and Cursor in the process. These tools provide auto-completion based on LLMs. We used Codex and Claude Code to help prove facts about association lists and heterogeneous lists, but all the theorems and lemmas are stated manually. We wrote the paper completely manually. We used AI tools inside Overleaf to help with editing (*e.g.,* fixing grammars, polish wording). We used Cursor to edit notations for consistency.

1 Introduction

Verifying existing software is hard. Existing software is typically developed in mainstream programming languages that are not amenable to verification. These software programs involve complicated optimizations that obscure the underlying logic. Their code is gigantic

in size. Even though there have been many verification tools for mainstream languages like C [3, 24, 26], Haskell [6, 29], and Rust [8, 11, 20, 23, 30], *etc.*, verifying existing code remains a daunting task.

How can we ease this task? Instead of directly reasoning about a complicated program, we can try to *simplify* it first. This approach has been pioneered by AutoCorres [15–17] in the context of C and Simpl [27, 28]. AutoCorres and its recent fork, AutoCorres2 [5], have been used in various frameworks, demonstrating their usefulness.

However, AutoCorres is based on a set of specific computation monads. This forces us to stick to one underlying semantic model. For example, AutoCorres/AutoCorres2 represent global variable read and write as state monad computation, which assumes that the global memory operations are thread-safe. In this paper, we explore that we can apply techniques underlying AutoCorres to more general models of computation.

In particular, interaction trees (ITrees) are a coinductive and executable model of computation [32]. They are parameterized over any compositions of effects, which allows them to model a variety of different language features. Unlike classical computation monads, the interpretation of these effects are delayed in ITrees, making it possible to switch effect semantics under different environments. In an ITree, we can leave global memory reads and writes uninterpreted, so that we can simplify a program first, then decide on the semantics model for global memory later. These characteristics also make ITrees an ideal language-agnostic interface for both imperative and functional languages. In this paper, we show how to implement *local variable lifting* (LVL), a key idea underlying AutoCorres, for ITrees while keeping the modularity promise.

We make the following contributions:

- We build a semi-automatic and proof-guided LVL framework for ITrees in Rocq. Our approach allows us to obtain a simplified program and a correctness proof of the simplification at the same time. Our approach supports reasoning about conditionals and loops based on three predicates (Section 4).
- We conduct a qualitative case study based on three small, simple, but illustrative examples. We show that our implementation produces similar simplified code to code generated by AutoCorres2. We also show our techniques can effectively simplify ITrees reasoning by removing proof obligations regarding local memory (Section 5).

In addition, we demonstrate our approach using a motivating example in Section 2. We define an interface for languages with local variables for implementing LVL based on ITrees in Section 3. Finally, we discuss related work in Section 6 and conclude in Section 7. All the Rocq formalization of this paper can be found in our artifact, which will be made publicly available upon acceptance.

2 Motivating Example

In this section, we illustrate our approach via a simple example presented in Figure 1. We start from a function `bump_counter` written in C (Figure 1a). The function takes an integer `n`. It runs for `n` iterations and increments a global counter `Counter` at each iteration. In order to illustrate multiple types in local variable lifting, we use a boolean `fg` to control when the loop stops, and write `false` to the flag when the counter is incremented `n` times. This function is simple, but its behavior is common in real-world applications, *e.g.,* profiling.

To reason about this program, we need to first "translate" it to a language amenable for verification. In this paper, we choose Rocq as the target language. Such a "translation" is shown in Figure 1b. On the surface, the Rocq function looks similar to the original

```
void bump_counter(int n) {
  int i = 0; bool fg = false;
  while (!fg) {
    fg = i >= n;
    if (!fg) {
      Counter += 1;
      i++;
    } else
      fg = true;
  }
}
```

(a) Function that takes an integer `n` and repeatedly increments a global variable `Counter` `n` times.

```
fun n : Z =>
iter (fun (a, _): hlist [Z; bool] * unit =>
  if negb (hhd (htl a)) then
    if negb (hhd a >=? n) then
      c <- read Z "Counter";;
      write Z "Counter" (c + 1);;
      Ret (inl ([hhd a + 1;hhd a >=? n],tt))
    else
      Ret (inl ([hhd a;true],tt))
  else
    Ret (inr ([hhd a;hhd (htl a)],tt))
) ([0;false],tt);;
Ret tt
```

(c) The `bump_counter` function after local variable lifting, named `simple_bump_counter`.

```
fun n : Z =>
write Z "n" n;;
write Z "i" 0;;
write bool "fg" false;;
iter (fun _ : unit =>
  vflag <- read bool "fg";;
  if negb vflag then (
    vi <- read Z "i";;
    vn <- read Z "n";;
    write bool "fg" (vi >=? vn);;
    b <- read bool "fg" ;;
    if (negb b) then (
      c <- read Z "Counter";;
      write Z "_t" c;;
      t <- read Z "_t";;
      write Z "Counter" (t + 1);;
      i' <- read Z "i";;
      write Z "i" (i' + 1);;
      Ret (inl tt)
    ) else (
      write bool "fg" true;;
      Ret (inl tt)
    )
  ) else (
    Ret (inr tt)
  )) tt;;
Ret tt.
```

(b) Initial denotation of `bump_counter` as an ITree, manually constructed.

Figure 1 Pipeline paired with initial and simplified ITrees for `bump_counter`.

function in C, except that it's more verbose: we use explicit `read`s and `write`s to represent memory operations that are implicit in the C code. These operations work on a memory that uses strings representing variable names, and these strings are indices into our memory representation. Note that this function has two "variables" named `n`: the function argument and the variable on memory indexed by `"n"`. We will use this convention for other variables as well. In the beginning of the function, we write the content in `n` to the memory via a `write` to initiate the function's call state. In addition to variables `"n"`, `"i"`, `"fg"`, and `"Counter"`, this function also uses a temporary variable `"_t"` to store the old value of `"Counter"` when we increment `"Counter"` by 1. In this paper, the examples are manually translated from CompCert's Clight [4] AST in a syntax-directed manner: `Sset` and `Sassign` become `write`; `Evar` and `Etempvar` become `read`; and `Ssequence`, `Sifthenelse`, and `Sloop` become `bind`, Rocq `if` expression, and `iter`, respectively.

This function is *monadic*: the `<-` and `;;` notations are like assignments and sequencing in C, but they are in fact monadic binds [25, 31]. We use an `iter` combinator to represent loops: the loop body inside `iter` returns a sum type; the loop continues if the returned type is a left injection (`inl`), and ends if it's a right injection (`inr`).

```
CoInductive itree (E : Type -> Type) (R : Type) : Type :=
| Ret (r : R)
| Tau (t : itree E R)
| Vis {A : Type} (e : E A) (k : A -> itree E R).
```

```
Variant MemoryE {K V} : Type -> Type :=
| Read  : ∀ (A : Type), `{Repr V A}
  K -> MemoryE A
| Write : ∀ (A : Type), `{Repr V A}
  K -> A -> MemoryE unit.
```

```
Variant ErrorE : Type -> Type :=
| Error : ∀ {A : Type},
  ErrorE A.
Definition E0 {E : Type -> Type} :=
  (MemoryE +' ErrorE +' E).
```

Figure 2 On the top, we show a simplified definition of interaction trees parameterized by an event `E`. On the bottom, we show two event interfaces. These definitions are simplified for presentation by excising implicit arguments. Here, we describe ITrees as a positive coinductive type following Rocq's historical style for simplicity, but ITree libraries define ITrees using negative coinductive forms [32]. This is because positive coinductive types are known to break subject reduction [13]. Constructors in this figure are implemented as "smart constructors" using notation.

We use Rocq's `Z` type to represent integers and `bool` types to represent booleans. We made a simplification here: we represent the bounded integer `int` in C using the unbounded integer `Z` in Rocq. Although this simplification will prevent us from reasoning about integer overflows, we can always change `Z` to a bounded integer type and our methodology still applies. In this paper, we only deal with unbounded integers for simplicity.

Interaction trees. The monad we use here is called *interaction trees*, or ITrees [32]. Unlike classical computation monads like state monads, `read` and `write` operations in interaction trees are *uninterpreted*. If we are going to reason about their semantics, we define their interpreters separately – this allows us to decouple effects and the semantics of effects in different languages or systems. Furthermore, ITrees are parameterized by these effects, enabling us to mix in different sets of effects depending on the source language. ITrees are *coinductive*, allowing us to encode non-terminating programs with them.

We show a simplified definition of ITrees in Figure 2. An ITree has type `itree E R`, where `E : Type -> Type` is an *event parameter* and `R : Type` represents the type of an ITree's result. The definition of `itree` contains three constructors:

1. `Ret r`, a pure result;
2. `Tau t`, a silent step, then continue with `t`;
3. `Vis e k`, which performs a visible event `e : E A` and continues with `k : A -> itree E R`.

ITrees are also known as *freer monads* [22]. We can define `ret` and monadic `bind` on them, regardless of what event parameter `E` is present [22, 32].

To describe programs like Figure 1b using ITrees, we need to define required *events*. We show two common event types we used in Figure 2. `MemoryE` describes memory events such as `Read` and `Write` over a variable-value model of memory. We use a `Repr` typeclass to encode association between the source language (C here) types and Rocq types; we defer detailed discussion of `Repr` to Section 3. `ErrorE` describes runtime errors. Events can be composed like a sum type using the `+'` operator. For example, we define a special event transformer `E0` that combines `MemoryE` and `ErrorE` with another arbitrary event `E`.

We can "embed" an event in an ITree by applying the `Vis` constructor. Since this is a common pattern, the ITree library defines the following helper function:

```
Definition trigger {E : Type -> Type} : forall A, E A -> itree E A :=
  fun _ e => Vis e (fun x => Ret x).
```

As a convention, we use terms starting with capital letters such as `Read`, `Write` and `Error`, to represent events. We use `read`, `write` and `error` to represent the smart constructors of these events defined using `trigger`.

We say our representations based on ITrees are *language agnostic* because they are not tied to the abstract syntax tree or the semantics of a specific language. For example, if we switch the source language to Rust, we can still use ITrees. If these languages use different memory models, we only need to define a different interpreter for `read` and `write` operations. If these languages support additional features, we can mix in additional events in ITrees.

We can also implement loops using the `iter` combinator, whose type is:

```
iter : ∀ {E : Type -> Type} {R I : Type}, (I -> itree E (I + R)) -> I -> itree E R.
```

The `iter` combinator models the Elgot iteration [1, 14]. It takes a function representing a loop body (`I -> itree E (I + R)`) and an initial argument of type `I`. Operationally, the `iter` combinator repeatedly applies the loop body, starting with the initial argument. After every iteration, the loop body either returns a value of type `I`, indicating that the loop should continue, or a value of type `E`, indicating the end of the loop.

Local variable lifting. The representation in Figure 1b is complicated. Instead of directly reasoning about it, we want to first have it *simplified*.

But how should we do that? First, we observe that `"i"` and `"fg"` are local variables that cannot be modified by other functions. Instead of representing operations over them as uninterpreted `Read`s and `Write`s over memory, we can model them using Rocq variables. We can do this for `"n"` and `"_t"` as well, because `"n"` is passed into this function as a call-by-value argument and `"_t"` is a temporary variable that expires when the function call ends. This allows us to remove all the effectful `read`s and `write`s over them, so working with these variables is ultimately stateless. This process is known as *local variable lifting* [16] (LVL).

However, LVL was proposed by David Greenaway et al. with an implementation in Isabelle/HOL based on the Simpl language and a specific computation monad. In this paper, we extend it and present a novel re-implementation of LVL for interaction trees in Rocq.

Simplifying the loop in Figure 1b is a key challenge. To work with loops, we developed a mechanical approach based on different characteristics of local variables. Intuitively, we first observe that `"n"` is never written to in the loop, so we can treat it as a value. We then notice that `"_t"` is always first written to before anyone reads it, so we can remove the first `write` and replace all subsequent `read`s with `"_t"`'s initial value. After removing `"_t"`, we realize `"i"` and `"fg"` are the only remaining local variables that the loop body writes to, so the loop only needs to carry them in every iteration.

We show the simplified Rocq function in Figure 1c. This function only contains `read` and `write` operations on the global variable `"Counter"`. Variables `"n"`, `"i"`, and `"fg"` in the memory are directly represented using Rocq variables `n`, `hhd a`, and `hhd (htl a)`. The loop carries a heterogeneous list `a` (`hlist`) that contains all variables "participating" in the loop.

Proof-guided simplification. We do not wish to manually write the simplified code. Instead, we want the following characteristics in simplification: (1) We obtain the simplified code semi-automatically; (2) We can define customized simplification "recipes"; (3) We get the simplified code and a correctness proof of the simplification at the same time. In addition, ITrees pose a challenge: they are too dynamic to be pattern matched on. More concretely, the function continuation in the `Vis` constructor of ITrees (Figure 2) makes it impossible to define a function that simplifies an ITree.

```
Derive simple_bump_counter in (
  ∀ (n : Z),
    interp_local ["n"; "i"; "fg"; "_t"]
      (bump_counter n) ≈
    simple_bump_counter n
) as simple_bump_correct.
Proof. start_function. repeat forward.
  lvl_iter ["n"]
    (env["_t"] : TCtx [Z]) ["_t"]
    (env["i"; "fg"] : TCtx [Z; bool]).
   repeat forward. forward_if.
  (* A series of similar tactics omitted. *)
  finish.
Defined.
```

```
∀ rem (i : Z) s,
  i + Z.of_nat rem = Z.max i n ->
  get_memory s "n" = Some (Vint n) ->
  get_memory s "i" = Some (Vint i) ->
  get_memory s "fg"
    = Some (Vbool false) ->
  iter body_o (s, tt);; Ret tt ≈
  bump_counter_trace rem
```

```
∀ rem i,
  i + Z.of_nat rem = Z.max i n ->
  iter body_l ([i; false], tt);;
  Ret tt ≈
  bump_counter_trace rem
```

Figure 3 On the left, we show the `Derive` command that simplifies `bump_counter` to a locally lifted form. This proof is simplified to omit some extraneous arguments. On the right, we show a comparison of loop invariants for `bump_counter` and `simple_bump_counter`, with the unlifted ITree invariant on the top, and the locally lifted ITree invariant on the bottom.

Driven by these goals and challenge, we develop a proof-guided simplification process to implement local variable lifting using Rocq's `Derive` extension.[1] `Derive` allows us to meta-program Rocq terms in the style of Bird-Meertens program derivations [12], *i.e.,* we write a proposition that relates an existing Rocq term to an existential variable, then use a constructive proof to instantiate the existential variable.

We show how we use the `Derive` command to obtain the simplified code of `bump_counter` in Figure 3. The command declares a new existential variable, `simple_bump_counter`. The proposition followed by `in` describes that for locals `"n"`, `"i"`, `"fg"`, and `"_t"`, interpreting local memory events in `bump_counter` (`interp_local`) produces an ITree equivalent to `simple_bump_counter`, according to an observational equivalence relation ≈ over ITrees. We only interpret local memory effects in `bump_counter`. We do *not* need to interpret these effects in `simple_bump_counter` because all local memory effects are lifted. We name this proposition `simple_bump_correct`.

Our method does not rely on the `Derive` extension. Indeed, we can implement our proof-guided simplification process without the `Derive` extension, *e.g.,* by using proof mode to define a term of a sigma type. However, the `Derive` command makes the process simpler.

After the `Derive` command, we can enter Rocq proof mode to prove this proposition. We have defined a number of proof tactics that implement local variable lifting that help us obtain `simple_bump_counter`. For programs without conditionals or loops, we can simplify an ITree by interpreting them over a local memory, similar to normalization by evaluation [9]. For programs with conditionals or loops like `bump_counter`, we need to apply all the observations described earlier. We have defined three predicates `NoWrites`, `WriteFirst`, and `WriteWithin` to capture characteristics of local variables. Applying our tactics require the user to specify for which variables the predicates hold.

In the `simple_bump_correct` example, we use `start_function` to set up our proof over a function, then repeatedly take evaluation steps `forward` that simplify the three `write` events in the beginning of Figure 1b, reaching `iter`. To lift `iter`, we call our `lvl_iter` tactic,

[1] `https://rocq-prover.org/doc/v9.0/refman/addendum/miscellaneous-extensions.html`

with arguments establishing `"n"` as `NoWrite`, `"_t"` as `WriteFirst`, and both `"i"` and `"fg"` as `WriteWithin` in the loop. We defer detailed definitions of these predicates to Section 4. The rest of the proof cleans up and repeatedly takes `forward` steps.

Notably, we can use simple proof tactics like `reflexivity` to obtain a naïve version of `simple_bump_counter` that is not simplified. Indeed, nothing in `simple_bump_correct` states that `simple_bump_counter` is simpler. In practice, we need to be mindful about how we construct proofs and apply our proof tactics so that we end up with simplified ITrees.

What do all these mean to proofs? Does the simplification help with proofs? In `bump_counter`, the answer is yes. First, we do not need to reason about `read`s and `write`s over local memories. Second, we obtain a simpler loop invariant on `bump_counter`.

To demonstrate this, we present loop invariants for `bump_counter` and `simple_bump_counter` to be $\approx$-equivalent to the specification `bump_counter_trace`: It specifies that for an input n, the program generates a trace repeating n `read`s and `write`s of `"Counter"` with correct values. Figure 3 shows the loop invariants of both programs in the right column. Each invariant specifies the loop body passed to `iter`, where `body_o` and `body_l` are from Figure 1b and Figure 1c, respectively. Both loop invariants describe the relation between `i`, `n`, and the number of iterations remain. They also maintain a relation between the loop and the specification. However, the loop invariant for `bump_counter` (upper-right) needs to additionally maintain knowledge about three key-value pairs in the memory state `s`. In contrast, the loop invariant for `simple_bump_counter` (lower-right) is stateless. The loop body only maintain an `hlist` that includes the exact values participating in this loop.

LVL does not change what is true; it changes *where that truth lives*. Without LVL, all information about local variables is buried in memory, so we must manually expose it as hypotheses and repeatedly prove that memory operations preserve it. With LVL, local variables participating in a loop are explicit and loop invariants are stated directly over values.

3 Language Definition

In this section, we discuss our basic framework. Our goal is to extend local variable lifting (LVL) to interaction trees. To demonstrate that, we have built a small language in ITrees that can model local memory of a variety of languages.

Events and memory model. We use events `MemoryE` and `ErrorE` shown in Figure 2 for our small language. `MemoryE` events parameterize over arbitrary variable types `K` and value types `A`. `Read A k` represents reading a variable `k` from the memory. This read expects a value of type `A`. `Write A k v` represents writing a value `v` of type `A` to variable `k`. When we handle these events, we only interpret `MemoryE` events over local variables while leaving `MemoryE` events over global variables and all other events uninterpreted.

We model the memory using an association list as a key-value map, where variable names (*i.e.,* strings) are keys. To store values of different types in the memory, we use a closed family of tagged types representing in-memory values. We use a typeclass `Repr V A`, shown in Figure 4a, to correspond this closed family of types `V` with Rocq types `A`. The methods `inj` and `prj` represent injection to produce in-memory values and projection out of in-memory values to untagged external values, respectively.

```
1  Definition handle_Write `{Repr V A} (locals : list K) (k : K) (a : A) (st : Memory)
2    : itree E0 (Memory * unit) :=
3    if var_in locals k then Ret (set_memory st k (inj a), tt)
4    else write A k a ;; Ret (st, tt).
5  Definition handle_Read `{Repr V A} (locals : list K) (k : K) (st : Memory)
6    : itree E0 (Memory * A) :=
7    if var_in locals k then read_state k st
8    else v <- read A k ;; Ret (st, v).
9  Definition handle_memory (locals : list K)
10   : ∀ T : Type, MemoryE T -> stateT Memory (itree E0) T :=
11   fun _ e st => match e with
12                 | Read k    => handle_Read  locals k   st
13                 | Write k a => handle_Write locals k a st
14                 end.
15 Definition handle_local (locals : list K)
16   : ∀ T : Type, E0 T -> stateT Memory (itree E0) T :=
17   fun R e st => match e with
18                 | inl1 el => handle_memory locals _ el st
19                 | inr1 er => (* omitted, leaves other events uninterpreted *)
20                 end.
21 Definition interp_local (locals : list K) (t : itree E0 R) : itree E0 R :=
22   map snd (interp_state (handle_local locals) t empty_memory).
```

```
1  Class Repr {V : Type} (A : Type) :=
2  {
3    inj : A -> V;
4    prj : V -> option A;
5    prj_inj :
6      ∀ a, prj (inj a) = Some a;
7  }.
```

(a) The definition of the `Repr` typeclass.

```
1  Definition read_state `{Repr V A}
2             (st : Memory) (k : K)
3    : itree E0 (Memory * A) :=
4    match get_memory st k with
5    | Some v => match prj v with
6                | Some a => Ret (st, a)
7                | None => error st
8                end.
9    | None => error st
10   end.
```

(b) The definition of the `read_state`.

Figure 4 Definitions related to our interpretation and memory model, simplified for presentation.

Handlers. We interpret events by defining their *handlers*. Figure 4 gives handlers for `MemoryE` and `E0`. Given an event e : `E0`, the handler `handle_local` (lines 15–20) dispatches e to `handle_memory` or leaves it uninterpreted, depending on whether e is a `MemoryE` event or not; and transforms `E0 T` to `stateT Memory (itree E0) T`, where `stateT` is the *state monad transformer* [21]. Memory events `MemoryE` are handled by `handle_memory` (lines 9–14): on a `Read` or `Write` of variable `k`, it checks whether `k` is local: if so, we handle `Write` by updating the `Memory` with `set_memory`, and we handle `Read` via `read_state` (Figure 4b), which looks up `k` with `get_memory` and projects the result, raising `Error` on a missing variable or failed projection. If `k` is not local, it re-emits the `MemoryE` event with its smart constructor. The top-level function `interp_local` passes these handlers to a helper function from the ITrees library `interp_state`, which mediates the application of our handlers and threading of state. The function `interp_local` then *drops* local memory from the result using `map`, where the `map` function is the standard functor map defined on the ITree monad.

Challenges of reasoning about interpretation. We can now reason about programs represented in ITrees using our interpreters. Given a set of local variables L, a state s, and an ITree t of type `itree E0 R`, we use $[\![t]\!](s)$ to denote `interp_state (handle_local` L`)` t s. The set of local variables L is left implicit in the notation $[\![t]\!](s)$; throughout the rest of the paper, it is the L fixed by the surrounding context. We write $s[x := v]$ to denote `set_memory s x v`, and `{ x:= vx; y:= vy; ... }` for the memory obtained by setting `x` to `vx`, `y` to `vy`, and so on; and we write $\emptyset$ for the empty memory.

We have the following interpretation rules for local memory events: for any variable x in given a set of local variables L, we have $[\![\texttt{read A } x]\!](s) \approx \texttt{read\_state } s\ x$ and $[\![\texttt{write A } x\ v]\!](s) \approx \texttt{Ret } (s[x := \texttt{inj } v], \texttt{tt})$. A local write is total: it updates the `Memory` by storing the injected value at the given variable. In contrast, reducing `read` to a `Ret` requires two facts: the variable is present in the current `Memory` (Figure 4b, line 5) and the stored value successfully projects from `V` to the requested type `A` (Figure 4b, line 6).

For straight-line code, we can reduce a `read` to a `Ret` by evaluating the preceding `write`s. Consider a `write` immediately followed by a `read` of the same variable:

$$(s', \_) \leftarrow [\![\texttt{write Z "x" 42}]\!](s)\ ;;\ [\![\texttt{read Z "x"}]\!](s') \approx [\![\texttt{read Z "x"}]\!](s[\texttt{"x"} := \texttt{Vint 42}]). \quad (1)$$

The left-hand side simplifies to the right-hand side by interpreting `write` and then applying the monad laws. Any subsequent `read` will be reading from a concrete state $s[\texttt{"x"} := \texttt{Vint 42}]$, where the existence of `"x"` is immediate. Moreover, `"x"` is an integer (shown by its tag `Vint`), which is exactly what `read` expects, we can reduce `read_state` to `Ret`.

Unfortunately, doing the same thing in a loop body or after a conditional is nontrivial. Consider the `bump_counter` example: let t_{iter} denote the body of `iter` in Figure 1b (lines 6–26). Interpreting the three writes that precede the loop yields $s_0 =$ `{"n":= Vint n;"i":= Vint 0;"fg":= Vbool false}`. We are then left to interpret `iter` t_{iter} `tt` from s_0. We do not want to interpret a loop, as a loop can be nonterminating. Instead, we want to inspect the loop body t_{iter}. But even if the loop body does not contain any additional loops, we still cannot know the values of local variables by interpreting operations on them. This is because the state in the loop body can be s_0, or the state after the first iteration, or any states after some iterations. There are also similar challenges to conditionals, because the join point of a conditional can come from either branch.

In larger programs, especially with nested conditionals and loops, the proof obligations for existence and type of local variables propagate through the bind structure. LVL is designed to make the relevant local-variable facts explicit at such control-flow boundaries.

4 Local Variable Lifting on ITrees

In this section, we present our implementation of local variable lifting (LVL) based on a proof-guided transformation on ITrees with local memory. LVL targets the two challenges identified last section. When s is universally quantified across iterations or differs across branches: reads of variables not written in the region cannot reduce without an invariant on s, while reads of written variables incur existence-and-type obligations across every bind. We address each with one ingredient. First, a *read-only interpretation* of `E0` – a handler parameterized by a fixed `Memory` containing bindings for unmodified variables, so their `read`s reduce to constants independently of the state threaded through the computation. Second, a set of *datatypes and operations* representing the post-state of a computation as Rocq variables – a typing context with a heterogeneous list of typed values – which propagates the existence and type of each lifted variable into every continuation. We then state theorems justifying LVL as semantics-preserving and discuss how the transformed ITrees ease verification.

```
Inductive TCtx {K V : Type}
  : list Type -> Type :=
| tctx_nil  : TCtx []
| tctx_cons :
    ∀ {A : Type} {ts : list Type}
      `{Repr V A} `{Default A},
      K -> TCtx ts -> TCtx (A :: ts).
```

```
Inductive hlist {V : Type}
  : list Type -> Type :=
| hnil  : hlist []
| hcons :
    ∀ {A : Type} {ts : list Type}
      `{Repr V A},
      A -> hlist ts -> hlist (A :: ts).
```

```
Definition extract_hlist : ∀ {ts}, TCtx ts -> Memory -> hlist ts.
Definition reconstruct_memory : ∀ {ts}, list K -> hlist ts -> Memory.
```

Figure 5 Typed environments describe lifted variables, while heterogeneous lists store their typed values. The implementation of `extract_hlist` and `reconstruct_memory` is omitted.

Read-only interpretation. The ITrees library lets us interpret `E0` through alternative handlers; we define the read-only handler with the following signature:

```
Definition handle_local_with_read_only (locals : list K) (read_only : Memory)
  : ∀ T : Type, E0 T -> stateT Memory (itree E0) T.
```

This handler behaves like `handle_local` (Figure 4, lines 15–20) on writes and on variables outside `read_only`. On a read, it first looks up the variable in `read_only` and only consults state if that lookup fails. Given a set of local variables L, two values r and s of the `Memory` type, and an ITree t, we write $[\![t]\!](r, s)$ for `interp_state` (`handle_local_with_read_only` L r) t s, leaving L implicit in the surrounding context. We call r the *read-only state*: it supplies immutable bindings used by the handler. We call s the *mutable state*, or simply the *state* when no ambiguity arises.

The key property is that the `read`s of variables from r reduce *independently* of s: $\forall x \in L$ and $a : A$, if `get_memory` r x = `Some` (`inj` a), then $[\![$`read A` $x]\!](r, s) \approx$ `Ret` (s, a). In `bump_counter`, binding `"n"` to its value n in r lets us reduce its `read`s inside the loop body to `Ret` (s, n) even when s is the abstract iteration state – exactly the step that `handle_local` could not take.

Datatypes and operations. In LVL, we define `TCtx`, a typing context for users to specify participating local variables, and `hlist`, a heterogeneous list of typed values. Our LVL tactics use this typing context to extract a heterogeneous list containing all values during proof-guided simplification.

We show all relevant definitions in Figure 5. First, a typing context uses a type-indexed data structure `TCtx`. `TCtx` stores *names* of all local variables in the data structure, and the associated type of each variable at the type level. For example, the typing context we specified in `bump_counter` (Figure 1b) is:

```
  tctx_cons "i" (tcxt_cons "fg" tctx_nil) : TCtx [Z; bool]
```

or `ctx["i"; "fg"] : TCtx [Z; bool]` with our custom notation. In addition, we require the indexed types of `TCtx` representable in memory via the `Repr` typeclass constraint. We enforce every indexed type to be inhabitable using a `Default` typeclass that can provide a "default value" of this type, similar to Breitner et al. [6]. Note that we can encode `TCtx` without using a list of `Type`s as its index, we choose to define it this way because it would make it easier to extract a heterogeneous list from it, as we will discuss shortly.

Common rules ($P \in \{\texttt{NoWrites}_L\ X,\ \texttt{WriteFirst}_L\ y,\ \texttt{WriteWithin}_{(\Gamma,L)}\ X\}$):

$$\frac{}{P\ (\texttt{Ret}\ s)}\text{PRet} \qquad \frac{P\ t}{P\ (\texttt{Tau}\ t)}\text{PTau} \qquad \frac{\forall v, P\ (k\ v) \qquad e \notin \{\texttt{Read A x}, \texttt{Write A x}\}}{P\ (\texttt{Vis}\ e\ k)}\text{POthers}$$

Specific rules:

$$\frac{\forall v, \texttt{NoWrites}_L\ X\ (k\ v)}{\texttt{NoWrites}_L\ X\ (\texttt{Vis}\ (\texttt{Read}\ A\ x)\ k)}\text{NWRead} \qquad \frac{x \notin L \vee x \notin X \qquad \forall v, \texttt{NoWrites}_L\ X\ (k\ v)}{\texttt{NoWrites}_L\ X\ (\texttt{Vis}\ (\texttt{Write}\ A\ x)\ k)}\text{NWWrite}$$

$$\frac{x \notin L \vee x \neq y \qquad \forall v, \texttt{WriteFirst}_L\ y\ (k\ v)}{\texttt{WriteFirst}_L\ y\ (\texttt{Vis}\ (\texttt{Read}\ A\ x)\ k)}\text{WFRead}$$

$$\frac{x \notin L \vee x \neq y \qquad \forall v, \texttt{WriteFirst}_L\ y\ (k\ v)}{\texttt{WriteFirst}_L\ y\ (\texttt{Vis}\ (\texttt{Write}\ A\ x)\ k)}\text{WFWriteNeq}$$

$$\frac{x \in L}{\texttt{WriteFirst}_L\ x\ (\texttt{Vis}\ (\texttt{Write}\ A\ x)\ k)}\text{WFWriteEq}$$

$$\frac{\forall v, \texttt{WriteWithin}_{(\Gamma,L)}\ X\ (k\ v)}{\texttt{WriteWithin}_{(\Gamma,L)}\ X\ (\texttt{Vis}\ (\texttt{Read}\ A\ x)\ k)}\text{WWRead}$$

$$\frac{x \notin L \vee (x \in X \wedge \Gamma \vdash x : A) \qquad \forall v, \texttt{WriteWithin}_{(\Gamma,L)}\ X\ (k\ v)}{\texttt{WriteWithin}_{(\Gamma,L)}\ X\ (\texttt{Vis}\ (\texttt{Write}\ A\ x)\ k)}\text{WWWrite}$$

Figure 6 Definitions of `NoWrites`, `WriteFirst`, and `WriteWithin`.

We then define a heterogeneous list `hlist` that contains all *values* of local variables. The `hlist` is also indexed by a list of `Type`s – this is a standard technique for encoding heterogeneous lists in dependently-typed languages like Rocq [7]. We implement two operations: `∀ t : Type, ts : list Type`, we get the first element of the `hlist` via `hhd : hlist (t :: ts) -> t`; and we get its tail via `htl : hlist (t :: ts) -> hlist ts`.

Given a typing context Γ : `TCtx ts` and a `Memory` s, `extract_hlist` Γ s extracts, for each variable named in Γ, its value from s and returns them in an `hlist` with the same type index `ts`: if the i-th entry of Γ names a variable `x` of type `A`, then the i-th entry of the `hlist` is the value of `x` and has type `A`. We call these variables the *lifted variables*. When `x` is absent from `Memory`, that slot in `hlist` is filled with the default value of `A` from the `Default` typeclass.

Conversely, `reconstruct_memory` builds a `Memory` from a list of variable names (usually from a typing context Γ) and an `hlist` of corresponding values, starting from an empty memory. Crucially, its implementation is *only* recursive on the `list` of variable names, not on `hlist`. It binds the i-th variable to `nth_inj` i hl, where `nth_inj : ∀ ts, nat -> hlist ts -> V` injects the i-th element of the `hlist` into `V` using the `inj` method supplied by the `Repr` typeclass. As a result, when `hl : hlist [Z; bool]` is universally quantified, `reconstruct_memory ["x"; "y"] hl` computes to `{"x":= nth_inj 0 hl;"y":= nth_inj 1 hl}`, where `nth_inj 0 hl` rewrites to `Vint (hhd hl)` and `nth_inj 1 hl` rewrites to `Vbool (hhd (htl hl))`. Thus the reconstructed memory has the same shape as the updated state in right-hand side of Term (1): each relevant variable is present, and each stored value is already an injection. Consequently, reading `"x"` or `"y"` with `read_state` reduces to `Ret` without any additional proof obligation.

Local variable lifting theorems. To perform LVL, we need knowledge of the *role* each local variable plays. We use three predicates to define these roles: (1) `NoWrites` describes an ITree where a specific set of local variables are *never* written to. (2) `WriteFirst` describes an ITree where a specific local variable is written *first* before being read. (3) `WriteWithin` describes an ITree where all writes are *within* a specific subset of local variables.

We define all three predicates coinductively in Rocq. We show these definitions as inference rules in Figure 6. All three predicates are parameterized by an ITree t : `itree E0 R` and a set of local variables L. Common rules PRET, PTAU, and POTHERS are shared across all predicates, as ITrees with no memory events naturally satisfy all these predicates.

$\texttt{NoWrites}_L\ X\ t$ asserts that t does not have any `Write` events to any variables in a set of variables X. The rule NWWRITE only allows `Write` to variables outside X or outside L.

$\texttt{WriteFirst}_L\ x\ t$ asserts that ITree t can never read the local variable `x` before it first writes to `x`, *i.e.,* the first local memory event involving `x` must be a `Write`. The rule WFREAD enforces that we can only read from variables that are not x, *i.e.,* no `Read` from x. Similarly, if there is `Write` on variables that are not x, we simply ignore this event by propagating the predicate, as described by WFWRITENEQ. Finally, the WFWRITEEQ specifies that an ITree that starts with a `Write` to x is `WriteFirst`.

The `WriteWithin` predicate is additionally parameterized by a typing context Γ : `TCtx ts`, where `ts` is a list of `Type`s. $\texttt{WriteWithin}_{(\Gamma,L)}\ X\ t$ asserts that every local `Write` performed by t is to a variable of a set X. In addition, the type associated to `Write` agrees with the type assigned by Γ, as indicated by the WWWRITE rule.

To facilitate proving an ITree satisfies one of these predicates, we prove compatibility theorems for all three predicates over common ITree operators (*e.g.,* `bind`, `iter`, `trigger`) and conditionals (*i.e.,* `if`) in Rocq.

We now show that each predicate over an ITree t enables us to rewrite t to a semantically-equivalent ITree via Lemmas 1–3:

Given a state s and an ITree t, the equivalence $[\![t]\!](s) \approx [\![t]\!](\emptyset, s)$ lets us rewrite the top-level `Derive` goal into one using the read-only handler, initially with an empty read-only state. The following operations then enable moving bindings between the read-only state and the mutable state while reasoning about different subtrees of t:

```
Definition build_read_only : list K -> Memory -> Memory.
Definition remove_vars : Memory -> list K -> Memory.
Definition overwrite : Memory -> Memory -> Memory.
```

Here, `build_read_only` $X\ s$ extracts the X bindings from s, `remove_vars` $s\ X$ removes them from s, and `overwrite` $s_1\ s_2$ merges two `Memory`s, preferring s_2 on conflicts. For $[\![t]\!](\emptyset, s)$, if t never `write`s to X, then the X bindings of s may be moved from the mutable state to the read-only state as $[\![t]\!](\texttt{build\_read\_only}\ s\ X, \texttt{remove\_vars}\ s\ X)$. The following lemma justifies this transformation in a more general setting. Rather than requiring the read-only state to be empty, it permits an existing r_0 to be *enlarged*; this is needed when rewriting a subtree of $[\![t]\!](r_0, \cdot)$ in which yet more variables are identified as read-only.

▶ **Lemma 1** (`NoWrites` justifies a read-only state). *Given a state s, a read-only state r_0, a set of local variables L, and an ITree t, if $\texttt{NoWrites}_L\ X\ t$ for some set of variables X defined in s, then letting $r = \texttt{build\_read\_only}\ s\ X$,*

$$[\![t]\!](r_0, s) \approx (s', a) \leftarrow [\![t]\!](\texttt{overwrite}\ r_0\ r, \texttt{remove\_vars}\ s\ X);;\ \texttt{Ret}\ (\texttt{overwrite}\ s'\ r, a).$$

To move variables between read-only and mutable state, we *enlarge* the read-only state with `overwrite` $r_0\ r$ and *reduce* the mutable state with `remove_vars` $s\ X$. Since the handler `handle_local_with_read_only` returns only the mutable state, we merge the read-only bindings back before handing the result to the continuation.

Given an ITree t, the top-level goal in `Derive` is over `interp_local` (Figure 4), which uses `map snd` $[\![t]\!](\emptyset)$ to discard the final `Memory`, since the lifetime of local variables ends. We introduce a relation I that relates two tuples of `Memory` $\times\ R$ for some type R:

$$I(X)\ (s_1, r_1)\ (s_2, r_2) \coloneqq r_1 = r_2 \wedge \forall x \notin X.\ \texttt{get\_memory}\ s_1\ x = \texttt{get\_memory}\ s_2\ x.$$

This relation allows two memories s_1 and s_2 to differ on a chosen set of variables X while requiring results r_1 and r_2 to be equal. ITrees library allows parameterizing a binary relation R over $\approx$-equivalence, denoted as $\approx_R$, to relate the results from two ITrees by R. Thus $t_1 \approx_{I(X)} t_2$ permits the resulting `Memory`s to differ on variables in X, while preserving the returned value that remains after local variables go out of scope.

▶ **Lemma 2** (`WriteFirst` justifies dropping variables). *Given a state s, a read-only state r, a set of local variables L, an ITree t : `itree` E_0 A, and its continuation over an ITree $k : A \to$ `ITree` E_0 B for some types A and B, if $\forall a : A,\ x \in X,$ `WriteFirst`$_L$ x $(k\ a)$ for some set of variables X, then we have*

$$(s', a) \leftarrow [\![t]\!](r, s)\ ;;\ [\![k\ a]\!](r, s') \approx_{I(X)} \tag{2}$$

$$(s', a) \leftarrow \texttt{map}\ \big(\lambda(s', a).\ (\texttt{remove\_vars}\ s'\ X, a)\big)\ [\![t]\!](r, s)\ ;;\ [\![k\ a]\!](r, s'). \tag{3}$$

Lemma 2 states that, if some local variables X are always first written to before being read in a continuation k over an ITree (k is also known as a KTree [32]), we can safely drop these variables in k. We implement this by `map`ping over the results of t via `remove_vars` s' X. Because `WriteFirst` is also satisfied by variables that are never read or written, this lemma holds only over $\approx_{I(X)}$. After X is dropped from the post-state of t, the continuation k may not `write` every variable in X, so the final states may still differ there.

Lemma 2 is especially useful with conditionals and loops. If one can assume their ITrees satisfy a scoping discipline common in modern programming languages: variables declared and initialized within one branch of a conditional, or within the body of a loop, are not meant to be used by the continuation after that control-flow construct. Such variables are therefore dead for the continuation, a condition captured by the `WriteFirst` predicate.

The purpose of dropping these bindings is to make local states compatible at the join point of a conditional or a loop. Given a state s, we define its *domain*, denoted $\mathrm{dom}(s)$, as the set of variable names present in s. At the end of a conditional, branches may leave behind different dead bindings, giving their post-states different domains. Dropping these bindings aligns the post-states to a common local-state domain. Similarly, for loops, dropping iteration-local bindings keeps the state domain the same across iterations.

After the dead variables have been dropped, the next lemma rewrites the head computation appearing in Term (3), *i.e.,* `map` $\big(\lambda(s', x).\ (\texttt{remove\_vars}\ s'\ X, x)\big)\ [\![t]\!](r, s)$. First, we define two operations on typing context: `vars` Γ returns a `list` of variable names for some typing context Γ; Γ_1`++`Γ_2 concatenates two typing contexts yielding `TCtx` $(ts_1$`++`$\,ts_2)$ for some Γ_1 : `TCtx` ts_1 and Γ_2 : `TCtx` ts_2. Next, given a state s and a typing context Γ, we say s *obeys* Γ if $\mathrm{dom}(s) =$ `vars` Γ and the value of each variable in s is well-typed under Γ. The key idea is that if s obeys Γ, then `extract_hlist` Γ s yields hl, a `hlist` carries values of *all* variables in s where i-th value of the `hlist` corresponds to i-th variable name of Γ, so `reconstruct_memory` (`vars` Γ) hl has the same bindings as s.

▶ **Lemma 3** (`WriteWithin` justifies extracting heterogeneous lists). *Given a state s, a read-only state r, a set of local variables L, an ITree t, and two typing contexts Γ_1 : `TCtx` ts_1 and Γ_2 : `TCtx` ts_2, if*

1. *`remove_vars` s (`vars` Γ_2) obeys Γ_1, and*
2. *`WriteWithin`$_{(\Gamma_1\texttt{++}\Gamma_2, L)}$ (`vars` $\Gamma_1 \cup$ `vars` Γ_2) t,*

then, for any ITree t, we have

$$\begin{aligned}&\texttt{map}\ \big(\lambda(s', a).\ (\texttt{remove\_vars}\ s'\ (\texttt{vars}\ \Gamma_2), a)\big)\ [\![t]\!](r, s) \approx \\ &(hl, a) \leftarrow \texttt{map}\ \big(\lambda(s', a).\ (\texttt{extract\_hlist}\ \Gamma_1\ (\texttt{remove\_vars}\ s'\ (\texttt{vars}\ \Gamma_2)), a)\big)\ [\![t]\!](r, s)\ ;; \\ &\texttt{Ret}\ (\texttt{reconstruct\_memory}\ (\texttt{vars}\ \Gamma_1)\ hl, a).\end{aligned}$$

Lemma 3 describes that if we can remove certain local variables from the result of evaluating an ITree t (via `remove_vars` s' (`vars` Γ_2)), we can also extract a heterogeneous list from the rest of the local memory (via `extract_hlist` Γ_1 (`remove_vars` s' (`vars` Γ_2))). Initially, `remove_vars` s (`vars` Γ_2) obeys Γ_1. During the execution of t, `WriteWithin` restricts every `write` to a variable in Γ_1`++`Γ_2 and requires the written value to be well-typed in that context. Hence any post-state s' of $[\![t]\!](r, s)$ has no bindings outside `vars` $\Gamma_1 \cup$`vars` Γ_2, and the bindings for variables in Γ_1 remain well-typed. After removing `vars` Γ_2 from s', the remaining state again obeys Γ_1, so we can safely extract a `hlist` from it according to Γ_1.

After interpreting t, we still need a `Memory` to pass to its continuation. We could use the post-state produced by interpreting t, *but we intentionally choose not to*: instead, we reconstruct a `Memory` from the resulting `hlist` according to Γ_1. This makes the heterogeneous list appear in the continuation. For example, imagine the continuation of a conditional is interpreted from `reconstruct_memory ["x"; "y"] hl` given arbitrary `hl : hlist [Z; bool]`, which reduces to `{"x":= Vint (hhd hl);"y":= Vbool (hhd (htl hl))}`. Thus, the continuation keeps its original `Memory`-passing type, while reads of lifted variables reduce to values supplied by the heterogeneous list.

Putting Lemmas 1–3 together, we show the following main theorem that justifies LVL:

▶ **Theorem 4** (Local Variable Lifting). *Given a state s, a read-only state r_0, a set of local variables L, two typing contexts Γ_1 : `TCtx` ts_1 and Γ_2 : `TCtx` ts_2, an ITree t : `itree E0` A, and its continuation $k : A \to$ `itree E0` B for some types A and B, if*

1. *`NoWrites`$_L$ X t for some set of variables X,*
2. *$\forall a : A,\ y \in$ `vars` Γ_2, `WriteFirst`$_L$ y $(k\ a)$,*
3. *`WriteWithin`$_{(\Gamma_1\texttt{++}\Gamma_2, L)}$ (`vars` Γ_1`++`Γ_2) t,*
4. *`remove_vars` s (`vars` $\Gamma_2 \cup$ `vars` X) obeys Γ_1*

let r be (`build_read_only` s X), we have

$$\begin{aligned}
&(s', a) \leftarrow [\![t]\!](r_0, s)\ ;;\ [\![k\ a]\!](r_0, s') \approx_{I(\texttt{vars}\ \Gamma_2)} \\
&(hl, a) \leftarrow \texttt{map}\ \big(\lambda(s', x).\ (\texttt{extract_hlist}\ \Gamma_1\ (\texttt{remove_vars}\ s'\ (\texttt{vars}\ \Gamma_2)), x)\big) \\
&\qquad\qquad [\![t]\!](\texttt{overwrite}\ r_0\ r, \texttt{remove_vars}\ s\ X)\ ;; \\
&[\![k\ a]\!](r_0, \texttt{overwrite}\ (\texttt{reconstruct_memory}\ (\texttt{vars}\ \Gamma_1)\ hl)\ r).
\end{aligned} \tag{4}$$

Theorem 4 essentially combines Lemmas 1–3 together. It lifts variables from the resulting state of the head, then reconstructs the `Memory` for the continuation. To see how Theorem 4 simplifies ITrees in practice, consider the following conditional, where `c : bool` is a Rocq variable, `"A"` and `"B"` are two global variables, all other variables are local:

```
1 if c then
2   a <- read Z "A" ;; write "t1" Z a ;; vt <- read Z "t1" ;; write "x" Z (vt + 1)
3 else
4   b <- read Z "B" ;; write "t2" Z b ;; vt <- read Z "t2" ;; write "y" Z (vt + 1) ;;
5 x <- read Z "x" ;; y <- read Z "y" ;; Ret (x + y)
```

We instantiate Theorem 4 with initial state $s_0 =$ `{"x":= Vint 42;"y":= Vint 7}`, empty read-only state r_0, $X = \emptyset$, two typing contexts $\Gamma_1 =$ `ctx["x"; "y"] : TCtx [Z; Z]` and $\Gamma_2 =$ `ctx["t1"; "t2"] : TCtx [Z; Z]`, and let $f(s', x) = \big($`extract_hlist` Γ_1 $s', x\big)$.

Let t_1 and t_2 denote the branches at line 2 and 4, respectively, let k denote the continuation of the conditional (line 5). With Term (4), we rewrite $[\![$`if c then` t_1 `else` $t_2]\!](\emptyset, s_0)$ to `map` f $\big([\![$`if c then` t_1 `else` $t_2]\!](\emptyset, s_0)\big)$. Since `map` commutes through conditional, we further rewrite it to `if c then map` f $[\![t_1]\!](\emptyset, s_0)$ `else map` f $[\![t_2]\!](\emptyset, s_0)$. Then we rewrite each branch with

interpretation rules and `hlist` extraction, they become `a` ← `read Z "A"` ;; `Ret` (`[a + 1; 7]`, `tt`) and `b` ← `read Z "B"` ;; `Ret` (`[42; b + 1]`, `tt`), respectively. We rewrite $\lambda(s', \_).\ [\![k\ \texttt{tt}]\!](\emptyset, s')$ to $\lambda(\texttt{hl}, \_).\ [\![k\ \texttt{tt}]\!](\emptyset, \texttt{reconstruct\_memory}\ (\texttt{vars}\ \Gamma_1)\ \texttt{hl})$. Under universally quantified `hl`, the mutable state can still be rewritten to `{"x":= Vint (hhd hl);"y":= Vint (hhd (htl hl))}`. So the continuation becomes $\lambda(\texttt{hl}, \_).$ `Ret (hhd hl + hhd (htl hl))` with interpretation rules, where the `read`s in k are reduced without extra proof obligations.

However, for loops, we need a different theorem. This is because the `iter` combinator is defined on an ITree *continuation* of type `I -> itree E (I + R)`. This means that there are two places where `Memory` needs to be lifted and reconstructed in an `iter`: the state passed into each loop iteration, and the state passed into the continuation after the loop.

▶ **Theorem 5** (Local Variable Lifting for `iter`). *Given a state s, a read-only state r_0, a set of local variables L, two typing contexts $\Gamma_1 : \mathit{TCtx}\ ts_1$ and $\Gamma_2 : \mathit{TCtx}\ ts_2$, the body of the `iter` over an ITree $t : I \to \mathit{itree}\ E0\ (I + A)$, an initial value of the `iter` $i_0 : I$, and its continuation $k : A \to \mathit{itree}\ E0\ B$ for some types A and B, if*

1. $\forall i : I,\ \mathit{NoWrites}_L\ X\ (t\ i)$ *for some set of variables X,*
2. $\forall i : I,\ a : A,\ y \in \mathit{vars}\ \Gamma_2,\ \mathit{WriteFirst}_L\ y\ (t\ i)$, *and* $\mathit{WriteFirst}_L\ y\ (k\ a)$,
3. $\forall i : I,\ \mathit{WriteWithin}_{(\Gamma_1 +\!\!+ \Gamma_2, L)}\ (\mathit{vars}\ \Gamma_1 +\!\!+ \mathit{vars}\ \Gamma_2)\ (t\ i)$,
4. $\mathit{remove\_vars}\ s\ (\mathit{vars}\ \Gamma_2 \cup \mathit{vars}\ X)$ *obeys* Γ_1

let $f : \mathit{Memory} \times (I + A) \to \mathit{hlist}\ ts \times (I + A)$ be

$$f(s, ir) := \begin{cases} \mathit{inl}\ (\mathit{extract\_hlist}\ \Gamma_1\ (\mathit{remove\_vars}\ (\mathit{vars}\ \Gamma_2)\ s), i), & \text{if } ir = \mathit{inl}\ i \\ \mathit{inr}\ (\mathit{extract\_hlist}\ \Gamma_1\ (\mathit{remove\_vars}\ (\mathit{vars}\ \Gamma_2)\ s), r), & \text{if } ir = \mathit{inr}\ r \end{cases}, \tag{5}$$

let r be $(\mathit{build\_read\_only}\ s\ X)$. We have:

$$\begin{aligned} &(s', a) \leftarrow [\![\mathit{iter}\ t\ i_0]\!](r_0, s)\ ;;\ [\![k\ a]\!](r_0, s') \approx_{I(\mathit{vars}\ \Gamma_2)} \\ &(hl, a) \leftarrow \\ &\quad \mathit{iter}\ \big(\lambda(hl, i).\ \mathit{map}\ f\ [\![t\ i]\!](\mathit{overwrite}\ r_0\ r, \mathit{reconstruct\_memory}\ (\mathit{vars}\ \Gamma_1)\ hl)\big) \\ &\qquad\quad \big(\mathit{extract\_hlist}\ \Gamma_1\ (\mathit{remove\_vars}\ (\mathit{vars}\ \Gamma_2 +\!\!+ X)\ s), i_0\big)\ ;; \\ &[\![k\ a]\!](r_0, \mathit{overwrite}\ (\mathit{reconstruct\_memory}\ (\mathit{vars}\ \Gamma_1)\ hl)\ r) \end{aligned} \tag{6}$$

We now apply Theorem 5 to the `bump_counter` ITree of Figure 1b, taking t_{iter} to be the `iter` body (lines 6–26). We specialize the theorem with the read-only set $X = \{\texttt{"n"}\}$, $\Gamma_1 =$ `ctx["i"; "fg"] : TCtx [Z; bool]`, $\Gamma_2 =$ `ctx["_t"] : TCtx [Z]`, an empty read-only state r_0, and the initial state $s_0 =$`{"n":= n;"i":= 0;"fg":= false}` produced by the three writes on lines 2–4 of Figure 1b. This rewrites $[\![\texttt{iter}\ t_{iter}\ \texttt{tt}]\!](\emptyset, s_0)$ into a loop that threads `hlist [Z; bool]` between iterations. The initial `hlist` is `[0; false]` extracted from the values of `"i"` and `"fg"` in s_0. Let $r =$`{ "n" := Vint n }`, the loop body becomes

$$\lambda(\texttt{hl}, \_).\ \texttt{map}\ f\ [\![t_{iter}\ \texttt{tt}]\!](r, \texttt{\{"i":= Vint (hhd hl);"fg":= Vbool (hhd (htl hl))\}}).$$

Now, we rewrite it with interpretation rules, the `read`s of `"i"`, `"fg"`, and `"n"` reduce to `Ret`s of `hhd hl`, `hhd (htl hl)`, and `n` respectively; also with `map` commuting conditionals and applying `hlist` extraction, it becomes exactly the `iter` in Figure 1c (lines 2–12). Then we rewrite its continuation $\lambda(s', \_).\ [\![\texttt{Ret tt}]\!](\emptyset, s')$ to:

$$\lambda(\texttt{hl}, \_).\ [\![\texttt{Ret tt}]\!](\emptyset, \texttt{\{"i":= Vint (hhd hl);"fg":= Vbool (hhd (htl hl));"n":= Vint n\}}).$$

For a more complicated continuation, the same reduction applies to any `read` of `"i"`, `"fg"`, or `"n"` that occurs before the continuation modifies that variable.

```
bump_counter' ?n ≡ do {
  whileLoop (λ(fg, i) s. fg = 0) (λ(fg, i).
    condition (λs. ¬ ?n < i)
      (do { modify (Counter_''_update (λa. a + 1)); return (0, i + 1) })
      (return (1, i)))
    (0, 0); skip }
```

Figure 7 The simplified `bump_counter` obtained using AutoCorres2. We removed the guards Autocorres2 inserts to maintain bounded integers in order to better illustrate the similarities of our works on unbounded integers.

Limitations. Our `WriteFirst` rewrite can drop variables between loop iterations, but we also require these variables to be dead at the entry of the continuation of the loop. This is conservative: if a variable is always overwritten before any read in each iteration, then the loop-carried value is irrelevant even when the continuation reads the variable after the loop. Strengthening the rule to support this case is left to future work. Users must currently supply the variable sets for `NoWrites`, `WriteFirst`, and `WriteWithin`. In future work, we plan to integrate a liveness analysis to infer these sets automatically and reduce manual effort.

5 Case Study

To evaluate how our implementation works in practice, we conduct a qualitative case study based on three simple examples: `mult_by_add`, the running example documented in the AutoCorres2 manual; `bump_counter`, shown in Section 2; and `profiler_reset`, a function adapted from the seL4 microkernel that resets an array (or a block of memory) using a loop. We only present a full description for `bump_counter` here, and worked out examples for `mult_by_add` and `profiler_reset` are in the artifact.

We use these programs to evaluate our implementation along two dimensions:

- **Comparison with AutoCorres2:** Whether our local-variable lifting produces an embedding that matches AutoCorres2's treatment of lifting local variables.
- **Proof impact:** How local variable lifting affects formal reasoning effort.

Correspondence with AutoCorres2. For variables that must be carried across loop iterations (or more generally, across compound statements), AutoCorres2 returns these variables as a tuple. In contrast, our implementation returns a heterogeneous list.

We illustrate this correspondence using `bump_counter`. Figure 7 shows the AutoCorres2 output, and Figure 1c shows our simplification. In this example, the loop carries exactly two local variables between iterations: `i` and `fg`. In our embedding, these appear as the two elements of the `hlist` bound to `a`: the first element of `a` corresponds to `i` in AutoCorres2, and the second element corresponds to `fg`.

Across all three case studies, the number of lifted variables in our `hlist` matches the arity of the tuple returned by AutoCorres2. After lifting, the loop iterations communicate exclusively through the explicit `hlist` without relying on abstract local memories.

Differences with AutoCorres2. AutoCorres2 uses a specific computation monad to represent effects like mutable states, exceptions, and control flows. In contrast, we use interaction trees where effects are parameterized and their interpretation is delayed. As a result, our simplified code and AutoCorres2 differ in how they represent global states and control flow.

AutoCorres2 represents reads and writes of global states as operations on a state monad. We show `bump_counter` after local variable lifting in AutoCorres2 in Figure 7. State updates are expressed via `modify`, which takes a functional update (a function mapping the old state to a new state). In contrast, our implementation keeps global-memory effects *uninterpreted* as events. We also use an explicit read-compute-write pattern, *e.g.,* `x <- read A ... ;; write A ...`, as shown in Figure 1c. Because our implementation does not interpret operations over global states, we can plug in any interpretations later. This can be especially important for concurrent programs, where a global state might be managed by multiple threads/processes.

AutoCorres2 uses loop constructs (`whileLoop` in Figure 7) that include conditionals as termination guards. In contrast, we rely on the `iter` combinator provided by ITrees that implements Elgot iteration [1, 14]. In Figure 1c, the guard at line 4 implies that `hhd a >=? n` is `false` at line 7, but we do not have this simplification. AutoCorres2 does: in Figure 7, line 4 returns `0` as the value of `fg`, *i.e.,* Boolean false.

Proof impact of local variable lifting on ITrees. Local variable lifting significantly changes proof obligations for establishing *local variable properties*, an advantage we share with Autocorres2. To demonstrate this, we prove the same properties for each example based on ITrees with and without local variable lifting.

We have described how `simple_bump_counter` is maintains a simpler loop invariant than `bump_counter` in Section 2. This applies to the other two examples as well. For unlifted ITrees, we required administrative proofs about `get_memory` and loop invariants that bookkeeps the existence and types of loop carried variables in the state. After local variable lifting, we no longer need those administrative proofs and the loop invariants are simpler. Moreover, the lifted ITrees are structurally simpler. Interested readers can find more details in our artifact.

6 Related Work

Local variable lifting. LVL was initially proposed by Greenaway *et al.* as a key simplification step in AutoCorres [15–17]. AutoCorres is a tool that translates deeply embedded C terms to shallowly embedded, monadic terms in the Isabelle theorem prover. In addition to local variable lifting, AutoCorres also performs simplifications such as peephole optimization, exception elimination, and heap abstraction Recently, Brecknell *et al.* created a new fork of AutoCorres named AutoCorres2 [5] that is more efficient and supports additional C features like unions, function pointers, a limited set of gotos, and pointers to local variables.

Our work focuses on LVL for ITrees. Our implementation is based on the idea presented in Greenaway [15]. We compare our implementation with AutoCorres2 in the last section. We leave the implementation of other simplifications over ITrees as future work.

As described earlier, LVL in AutoCorres and AutoCorres2 is based on specific computation monads. To perform LVL, Greenaway first parses a monadic program into blocks and identifies a write set, a read set, an entry live set, and an exit live set of each block. These sets are then used to guide LVL. Our implementation does not include an automatic live variable analysis process, as we focus on implementing LVL for ITrees. We define three predicates on ITrees that roughly correspond to the four sets used by Greenaway: `NoWrites` corresponds to the read set minus the write set; `WriteWithin` corresponds to the write set; and `WriteFirst` corresponds to the *complement* of the entry live set, and the *complement* of `WriteFirst` corresponds to the exit live set.

LVL and the translation proof in AutoCorres and AutoCorres2 are fully automatic. The soundness of translation theorem states a refinement relation that there always exist a pair of executions between an unlifted and a lifted program that starts from the same precondition and corresponding states, and return the same value. Because the parsing and live variable analysis steps are not verified, the translation can fail to produce evidence for the refinement relation and not produce a translation. In contrast, LVL is proof-guided in our implementation. Our approach does not reason about sets of executions. Instead, we rely on an observational equivalence relation defined on ITrees. Our proof is only semi-automatic at this point, but we believe a syntax-guided fully automatic proof tactic should be possible.

Functional extraction. There are other works that extract functional abstraction from imperative programs, but do not rely on LVL. For example, Heapster uses a relational type system with separation logic [19]. Users write types describing memory properties such as pointer access permissions and memory shapes. From the types and the program, Heapster creates an ITree representation of the program, type-checks the program under their relational type system, and refines it to a functional specification without pointers. Heapster focuses on heaps and separation logic, while our work focuses on local variables in states. Aeneas [20] translates Rust (via MIR) into pure Gallina, leveraging ownership and borrow checking to ensure memory safety and enable functional extraction. Electrolysis [30] targets Lean, mapping immutable variables to pure values and modeling mutability with lenses. Both systems face a join problem at conditionals: Aeneas requires no code after `if`-statements in the source code, while Electrolysis duplicates the post-conditional continuation into each branch rather than computing a join state. The `Why3` [10] platform used by Creusot [8] also has a constrained specification language `WhyML`. `WhyML` constrains input programs to have a finite statically known set of names for l-values and disallows higher-order functions, so that `WhyML` programs can be represented purely.

Appel observes that Static Single-Assignment (SSA) is functional programming and suggests what SSA and functional programming can learn from each other [2]. We would like to explore this connection further in our future work.

Verified Constructions. Gruetter *et al.* propose a novel approach for developing correct-by-construction C programs [18]. Their work and ours intersect in providing an interface for users to construct both an imperative program and a specification of its behavior in Rocq simultaneously. We both use Ltac tactics to construct existential terms. We both include a "step" tactic that takes symbolic steps on the program. The approach of Gruetter *et al.* is tuned for proving C programs. It provides a set of tactics that are disguised as C syntax and comments, and requires the user to annotate the code with separation logic predicates.

7 Conclusion

In this paper, we present an implementation of LVL for interaction trees in Rocq. Our implementation is semi-automatic and proof-guided based on Rocq's `Derive` extension. We define three predicates, namely `NoWrites`, `WriteFirst`, and `WriteWithin`, that we can use to identify local variables that can be lifted, including inside conditionals and loops. We prove theorems based on these predicates that allow us to simplify programs. Finally, we conduct a case study based on three simple examples. We show that our implementation produces similarly simplified code to the code generated by AutoCorres. We also show that local variable lifting simplifies proof obligations for reasoning about ITrees.

References

1 Jirí Adámek, Stefan Milius, and Jirí Velebil. Elgot theories: a new perspective on the equational properties of iteration. *Math. Struct. Comput. Sci.*, 21(2):417–480, 2011. doi:10.1017/S0960129510000496.

2 Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998. doi:10.1145/278283.278285.

3 Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, United States, 2014. doi:10.1017/CBO9781107256552.

4 Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the C language. *CoRR*, abs/0901.3619, 2009. doi:10.48550/arXiv.0901.3619.

5 Matthew Brecknell, David Greenaway, Johannes Hölzl, Fabian Immler, Gerwin Klein, Rafal Kolanski, Japheth Lim, Michael Norrish, Norbert Schirmer, Salomon Sickert, Thomas Sewell, Harvey Tuch, and Simon Wimmer. AutoCorres2. *Arch. Formal Proofs*, 2024. URL: https://www.isa-afp.org/entries/AutoCorres2.html.

6 Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua M. Cohen, and Stephanie Weirich. Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code. *J. Funct. Program.*, 31:e5, 2021. doi:10.1017/S0956796820000283.

7 Adam Chlipala. *Certified Programming with Dependent Types – A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

8 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A Foundry for the Deductive Verification of Rust Programs. In Adrián Riesco and Min Zhang, editors, *Formal Methods and Software Engineering – 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*, volume 13478 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2022. doi:10.1007/978-3-031-17244-1_6.

9 Andrzej Filinski. Normalization by Evaluation for the Computational Lambda-Calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2001. doi:10.1007/3-540-45413-6_15.

10 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems – 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, Lecture Notes in Computer Science, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

11 Formal Land development team. rocq-of-rust. URL: https://github.com/formal-land/rocq-of-rust.

12 Jeremy Gibbons. The School of Squiggol – A History of the Bird-Meertens Formalism. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Creissac Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops – Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*, volume 12233 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 2019. doi:10.1007/978-3-030-54997-8_2.

13 Eduardo Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, École normale supérieure de Lyon, 1996. Thèse de doctorat dirigée par Bougé, Luc Sciences appliquées École normale supérieure de Lyon (Lyon ; 1987-2009) 1996. URL: http://www.theses.fr/1996ENSL0044.

14 Sergey Goncharov. Shades of Iteration: From Elgot to Kleene. In Alexandre Madeira and Manuel A. Martins, editors, *Recent Trends in Algebraic Development Techniques – 26th IFIP WG 1.3 International Workshop, WADT 2022, Aveiro, Portugal, June 28-30, 2022, Revised Selected Papers*, volume 13710 of *Lecture Notes in Computer Science*, pages 100–120. Springer, 2022. doi:10.1007/978-3-031-43345-0_5.

15 David Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, University of New South Wales, Sydney, Australia, 2014. doi:10.26190/unsworks/17396.

16 David Greenaway, June Andronick, and Gerwin Klein. Bridging the Gap: Automatic Verified Abstraction of C. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving – Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2012. doi:10.1007/978-3-642-32347-8_8.

17 David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: formal verification of C code without the pain. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom – June 09–11, 2014*, pages 429–439. ACM, 2014. doi:10.1145/2594291.2594296.

18 Samuel Gruetter, Viktor Fukala, and Adam Chlipala. Live Verification in an Interactive Proof Assistant. *Proc. ACM Program. Lang.*, 8(PLDI):1535–1558, 2024. doi:10.1145/3656439.

19 Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. A type system for extracting functional specifications from memory-safe imperative programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi:10.1145/3485512.

20 Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP):711–741, 2022. doi:10.1145/3547647.

21 Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136, Berlin, Heidelberg, 1995. Springer. doi:10.1007/3-540-59451-5_4.

22 Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. doi:10.1145/2804302.2804319.

23 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.

24 William Mansky and Ke Du. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.*, 8(POPL):148–174, 2024. doi:10.1145/3632848.

25 Eugenio Moggi. Notions of Computation and Monads. *Inf. Comput.*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

26 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of C code with refined ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, PLDI 2021, pages 158–174, New York, NY, USA, 2021. ACM. doi:10.1145/3453483.3454036.

27 Norbert Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, Germany, 2006. URL: http://mediatum.ub.tum.de/mediatum/servlets/TUMDistributionServlet?id=mediaTUM_derivate_000000000002972.

28 Norbert Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Arch. Formal Proofs*, 2008, February 2008. , Formal proof development. URL: https://isa-afp.org/entries/Simpl.html.

29 Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 14–27. ACM, 2018. doi:10.1145/3167092.

30 Sebastian Ullrich. Simple Verification of Rust Programs via Functional Purification. Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, December 2016. Master's thesis, Institute for Program Structures and Data Organization (IPD). URL: `https://pp.ipd.kit.edu/uploads/publikationen/ullrich16masterarbeit.pdf`.

31 Philip Wadler. Comprehending Monads. *Math. Struct. Comput. Sci.*, 2(4):461–493, 1992. `doi:10.1017/S0960129500001560`.

32 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. `doi:10.1145/3371119`.