Layered Indexed Effects
Foundations and Applications of Effectful Dependent Typing

ASEEM RASTOGI, Microsoft Research, India
GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina
AYMERIC FROMHERZ, Microsoft Research & CMU, USA
TAHINA RAMANANANDRO, Microsoft Research, USA
NIKHIL SWAMY, Microsoft Research, USA

Programming and reasoning about effects in a functional programming language is a long-standing research problem. While monads have been a widely adopted common basis for effects in general purpose languages like Haskell, to improve modularity, compositionality, and precision of static reasoning, many refinements and alternatives have been developed, including algebraic effects, graded monads, parameterized monads, Hoare monads, Dijkstra monads, producers, polynomials, and combinations thereof, to only name a few.

To benefit from all these approaches, we propose a new language feature, *layered indexed effects*, and implement it as an extension to the F★ programming language. Using our feature, programmers can extend F★ with user-defined effect disciplines in all the aforementioned styles. Layered indexed effects provide a type-and-effect directed elaboration algorithm which allows programs to be developed in applicative syntax and to be reasoned about in a programmer-chosen abstraction, hiding the underlying semantic representations that justify a given effect-typing discipline. Being dependently typed, F★ supports developing both the semantic models and reasoning about client programs within the same framework.

We evaluate our work on four case studies: a generic framework for operation-indexed algebraic effects and handlers, together with the derivation of a Hoare-style program logic to reason about their stateful behavior; a new graded monad for stateful information flow control; a library of verified low-level message formatters structured as a parameterized monad upon a Hoare monad; and a hierarchy of effects mixing graded monads, Hoare monads, polynomials and continuations, to structure programs and proofs in a concurrent separation logic. Additionally, layered indexed effects have also simplified the core of F★, allowing us to derive basic Dijkstra monad constructions in F★ that were previously primitive. Given our positive experience, we speculate that other dependently typed languages might benefit from layered indexed effects too.

1 INTRODUCTION

Monads have been a remarkably effective way to structure and reason about programs with computational effects [Moggi 1989; Wadler 1992]. Over the years, a great many computational effects have been shown to be expressible as monads, including state, exceptions, continuations, parsing, printing, asynchrony, and many others besides. Still, monads are far from the final word on effectful functional programming, particularly when using multiple effects.

Aiming to improve modularity, compositionality and precision, various refinements and alternatives to monads have been proposed. For instance, Plotkin and Power [2003] propose algebraic effects as a generic semantic basis for effects, and many others [Bauer and Pretnar 2015; Brady 2013; Leijen 2017; Lindley et al. 2017; Plotkin and Pretnar 2009] have developed programming languages and libraries to program compositionally with algebraic effects and handlers. To support formal reasoning about effectful computations, others have proposed refining monads with various indexing structures, including parameterized monads [Atkey 2009], graded monads [Katsumata...
2014], Hoare monads [Nanevski et al. 2008], Dijkstra monads [Swamy et al. 2013], products [Tate 2013], and polymonads [Hicks et al. 2014].

While each of these frameworks offers attractive benefits over vanilla monads for both program-
ming and proving, and several languages support various subsets of them, no single framework
supports them all. What is missing is a shared structure which captures the commonalities of all
these constructions, e.g., something akin to Wadler’s classic monad interface and accompanying
“do-notation” that makes programming with monads practical. The main contribution of this paper,
layered indexed effects, is such a structure: it allows defining a wide range of indexed monads to
encode a variety of effect disciplines, while offering a direct programming style with automatic
inference and elaboration into custom effectful semantics. By embedding layered indexed effects
inside F* [Swamy et al. 2016], a dependently typed programming language and proof assistant,
user-defined effect disciplines can also be used to structure formal proofs.

1.1 Layered Indexed Effects: A First Example

For a first example, consider the indexed monad \( \text{gst} \) (\( a \):\( \text{Type} \)) (\( t \):\( \text{tag} \)), with \( \text{tag} = R | RW \) shown below,
defining either a reader or a read-write state monad, with actions to \( \text{read} \) and \( \text{write} \) the state.

```plaintext
let state = int
type tag = R | RW
let (⊕) t₀ t₁ = match (t₀, t₁) with | (R, R) → R | _ → RW
let gst (a:Type) (t:tag) = match t with R → (state → a) | RW → (state → a + state)
let return (x:a) : gst a R = λs → x
let bind t₀ t₁ (f:gst a t₀) (g: a → gst b t₁) : gst b (t₀ ⊕ t₁) = ...
let read : gst state R = λs → s
let write (s:state) : gst unit RW = λ_ → (), s
```

This structure is an instance of Katsumata’s (2014) graded monad—a monad-like structure indexed
by a monoid \( M \) whose \( \text{return} \) and \( \text{bind} \) are specified in the indices by the identity and operator of
the monoid \( M \). In this case, the monoid in question is (\( \text{tag} \), \( ⊕ \), \( R \)). The tag index on \( \text{gst} \) provides useful
static reasoning about stateful programs, notably, \( \text{gst} a R \) computations do not modify the state.
However, programming with \( \text{gst} \) isn’t trivial. Consider incrementing the state: one would write
it as \( \text{bind read} (\lambda x → \text{write} (x + 1)) \), while hoping for type inference to instantiate many missing
implicit arguments. If one were interested in graded monads only, one could develop libraries to
support elaboration and inference for them—but, the diversity of effect typing disciplines would
require a proliferation of ad hoc approaches to cater to their specific needs.

Automated elaboration, subsumption, and abstraction. Using layered indexed effects, we can turn
the \( \text{gst} \) monad into a user-defined effect, as shown below. Doing so introduces a new indexed
computation type \( \text{GST} \) corresponding to \( \text{gst} \). The computation type \( \text{GST} \) classifies a family of effects
represented by \( \text{gst} \)—in this case, \( \text{read} \) and \( \text{read-write stateful} \) computations. Actions in \( \text{gst} \) can be
reflected as actions in \( \text{GST} \), in the spirit of Filinski’s (1999) monadic reflection and layered monads.

```plaintext
effect { GST (a:Type) (t:tag) = gst a t with return; bind }
let read (): GST state R = GST?reflect (λ s → s)
let write (s:state): GST unit RW = GST?reflect (λ_ → (), s)
```

Having introduced the \( \text{GST} \) effect, \( \text{F}^* \) can automatically typecheck effectful terms written in direct
style, and internally elaborate them into the more verbose monadic syntax. For instance, the term
\( \text{write} \ (\text{read} (i + 1)) \) is inferred to have the computation type \( \text{GST unit RW} \) and internally elaborated into
the explicit monadic form shown earlier. Optionally, programmers can also specify a subsumption
relation for an effect, to enable transparently reuse of, say, reader computations (with tag \( R \) in

contexts expecting read-write computations; branching computations with different indices can also be implicitly lifted and composed. Further, effect definitions can be layered on top of each other, e.g., we could add an additional layer to represent exceptions on top of the GST effect. Finally, being embedded in F★’s dependent type theory, parametric polymorphism over the effects within an effect family is also easily supported, as is refining GST with Hoare-style specifications.

1.2 Contributions

F★, prior to our work, provided only various flavors of Dijkstra monads (a monad-indexed monad) to reason about effectful programs. While Dijkstra monads, like other Hoare-style specification mechanisms chosen in other dependently typed languages [Charguéraud 2011; Chlipala et al. 2009; Letan and Régis-Gianas 2020; Nanevski et al. 2008], are attractive for their generality, one size doesn’t always fit all: it can be easier to work with domain-specific abstractions rather than general-purpose ones. Layered indexed effects allow programmers to build abstractions of their choice. But that’s not all – they also simplify the foundations of F★. Indeed, one of our core contributions is to show that Dijkstra monads need no longer be a primitive notion in F★ and can instead be a user-defined monad-indexed layered effect.

Elaborating effects into a core calculus of dependent refinement types (§3). Explicitly Monadic F★, or EMF★, a core calculus for F★ defined by Ahman et al. [2017], no longer needs its monadic parts. Instead, we simplify the core of F★ to a subset of EMF★, which we call TotalF★, a pure type system with refinement types. Based on this smaller core, we design LIF★, a surface language with user-defined layered indexed effects, and a simple type-and-effect directed elaboration of LIF★ programs into TotalF★. Theorem 3.1 proves that the elaboration of LIF★ to TotalF★ is well-typed. We have also implemented layered indexed effects in F★ and used them to express several new effect typing disciplines enabling new applications—we present four case studies.1

Algebraic effect handlers and Dijkstra monads (§2.1, §2.2). We develop a library of generic algebraic effects and handlers, using a graded monad to track the operations a computation may invoke. The library supports generic handling and interpreting into a semantics of the user’s choosing. We show how to interpret heap-manipulating programs into both a computational semantics and a predicate transformer semantics, deriving a Dijkstra monad (via a construction due to Maillard et al. [2019]) for stateful algebraic effect programs, which seamlessly integrate with F★’s SMT-based automated proving engine.

Stateful information flow control (§2.3). We present a new graded monad for information flow control (IFC) in stateful computations. Our approach is more precise than prior approaches to monadic static information flow control and internalizes the proof of noninterference. Using the abstraction of layered indexed effects, client programs can be verified for noninterference while reasoning only on IFC label orderings, instead of relational proofs on the underlying semantics.

Binary message formatting (§4). We improve support for low-level message formatting in EverParse [Ramananandro et al. 2019], a library of monadic parser and formatter combinators. By layering a parameterized monad on top of EverParse’s existing use of a Hoare monad for C programs, we obtain more concise programs and better proof automation, while still supporting extraction to C code, with the abstraction of layered indexed effects imposing no runtime cost.

Concurrent separation logic (§5). Finally, we develop a hierarchy of effect layers including two graded Hoare polymonads, two non-graded Hoare polymonads, and two further effect layers for

1We submit our formal development and code for all the case studies as anonymous supplementary material.
capturing continuations, to structure and simplify programs and proofs in Steel [Swamy et al. 2020],
an embedding of concurrent separation logic in F∗.

The diversity of our experience encourages us to conclude that layered indexed effects are a
simple but powerful language mechanism which allows programmers to easily define layered
effect-typing disciplines in abstractions of their choosing, and enjoy a direct programming style
with automatic inference and elaboration into custom effectful semantics.

2 ENCODING EFFECT TYPING DISCIPLINES

This section introduces layered indexed effects and F∗ with three main examples. First, we review
Dijkstra monads and show how they no longer need to be primitive in F∗. Next, we develop a library
of generic algebraic effects and handlers and provide two typing disciplines for them: first, a graded
monad recording unhandled actions and then, a Dijkstra monad for verifying programs using a
mutable heap. Finally, we present an effect discipline for stateful information flow control. In all
cases, the abstraction and elaboration mechanisms offered by layered indexed effects allow client
programs to be written in native, direct-style syntax, and verified in a user-chosen abstraction,
hiding details of the underlying semantic representations.

As such, layered indexed effects generalizes F∗ both “upwards” and “downwards”: upwards, by
allowing more interesting indexing structures that were previously out of reach; and downwards,
by removing Dijkstra monads from its core, a welcome simplification for a theorem prover.

2.1 Introducing F∗, Dijkstra Monads, and Layered Indexed Effects

F∗ is a program verifier and a proof assistant based on a dependent type theory with a hierarchy
of predicative universes (like Coq or Agda). F∗ also has a dependently typed metaprogramming
system inspired by Lean and Idris (called Meta-F∗ [Martínez et al. 2019]) that allows using F∗ itself
to build and run tactics for constructing programs or proofs. More specific to F∗ is its effectful type
system, extensible with user-defined effects, and its use of SMT solving to automate some proofs.
To date, effects in F∗ have been tied to Dijkstra monads [Ahman et al. 2017; Swamy et al. 2013]
—no longer, as we will soon see.

Basic Syntax. F∗ syntax is roughly modeled on OCaml (val, let, match, etc.) though with differ-
ences due to the additional typing features. Binding occurrences b of variables take the form x:t,
declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The
syntax \( \lambda (b_1) \ldots (b_n) \rightarrow t \) introduces a lambda abstraction, whereas \( b_1 \rightarrow \ldots \rightarrow b_n \rightarrow c \) is the shape of a
curried function type. Refinement types are written \( b[t] \), e.g., \( \text{nat} = x:int[x \geq 0] \) represents natural
numbers. We define squash \( t \) as the unit refinement \( \_\text{unit}[t] \), which can be seen as the type of
(computationally irrelevant) proofs of \( t \). As usual we omit the type in a binding when it can be
inferred; and for non-dependent function types, we omit the variable name. For example, the type
of the append function on vectors is \( \#a:\text{Type} \rightarrow \#m:\text{nat} \rightarrow \#n: \text{nat} \rightarrow \text{vec } a \rightarrow \text{vec } a \rightarrow \text{vec } a (m+n) \),
where the two explicit arguments and the return type depend on the three implicit arguments
marked with ‘#’. We mostly omit implicit binders, except when needed for clarity, treating all
unbound variables in types as prenex quantified. The type of pairs in F∗ is represented by a \& b; in
contrast, dependent tuple types are written as \( x:a \& b \) with \( x \) bound in \( b \). A dependent pair value is
written (\( e, f \)) and we use \( x_1 \) and \( x_2 \) for the first and second dependent projection maps. We elide
universe annotations throughout this paper.
A Dijkstra monad is a monad-like structure \( M \) indexed by a separate monad of specifications \( W \),
forming types of the shape \( M a w \), where \( w : W a \). Dijkstra monads in F∗ have traditionally taken \( W \)
to be a weakest-precondition monad [Ahman et al. 2017; Swamy et al. 2016, 2013] in the style of
Dijkstra (hence their name), though this is not a fundamental requirement [Swamy et al. 2019].
A simple Dijkstra monad for pure computations. Consider the identity monad \( \text{id} \ t = t \), whose return is \( \lambda x \rightarrow x \) and whose bind is the (reverse) function application \( \lambda x \rightarrow f \ x \). A simple Dijkstra monad for \( \text{id} \) is an indexed monad, \( \text{pure} \ (t \cdot \text{Type}) \ (w : w \ t) \), where \( w \ t = (t \rightarrow \text{prop}) \rightarrow \text{prop} \) is the type of a predicate transformer for \( \text{id} \ t \) computations, transforming a precondition (a predicate on results, \( t \rightarrow \text{prop} \)) to a precondition (a proposition, \( \text{prop} \)). Notice that \( w \ t \) is itself a continuation monad (with answer type \( \text{prop} \)), with \( \text{wreturn} \ x = \lambda p \rightarrow p \ x \) and \( \text{wbind} \ w_1 \ w_2 = \lambda p \rightarrow w_1 (\lambda x \rightarrow w_2 \ p) \). As we will see shortly, the return and bind on pure computations manifest in the indices via \( \text{wreturn} \) and \( \text{wbind} \). This makes it natural to compute \( \wp \) specifications for pure computations, and to use those specifications for Hoare-style program proofs. For example, integer division can be given a type such as \( x : \text{int} \rightarrow y : \text{int} \rightarrow \text{pure} \int (\lambda p \rightarrow y \neq 0 \land p \ (x / y)) \), indicating a pure function on integers \( x \) and \( y \), with a precondition that \( y \) is non-zero and returning a value \( x/y \).

Computation types: \( \text{Tot} \), \( \text{PURE} \), and user-defined Dijkstra monads. Recognizing its generality and correspondence to weakest precondition calculi for Hoare logics, \( F^* \) baked in notions of Dijkstra monads. Towards that end, \( F^* \) contains a notion of a computation type, distinguishing computations from values in a manner similar (though not identical) to Levy [2004]. In their most general form, arrow types in \( F^* \) are written \( x : t \rightarrow C \) where \( C \) is a computation type, including \( \text{Tot} \ t \), for total computations\(^2\) (corresponding to the identity monad), and a variety of other Dijkstra monads all with the shape \( M \ t \ wp \), where \( wp \) is a monad at the level of specifications. Here \( M \) ranges over \( \text{PURE} \), the primitive Dijkstra monad for pure computations, as well as other user-defined Dijkstra monads, defined in terms of \( \text{PURE} \). Two core typing rules (shown slightly simplified) internalize the use of Dijkstra monads, allowing values to be returned in any computation type \( M \) (corresponding to a return in \( M \)'s index) and for \( M \)-computations to be sequenced (with a bind at the level of indices).

\[
\begin{align*}
\text{T-Return} & : \quad \Gamma \vdash v : \text{Tot} t \\
\text{T-Bind} & : \quad \Gamma \vdash e : M\ a\ w_1 \\
& \quad \Gamma, x : a \vdash f : M\ b\ (w_2\ x) \\
\Gamma \vdash \text{let}\ x = e\ \text{in}\ f : M\ b\ (M\ \text{bind}\ w_1\ w_2)
\end{align*}
\]

The shared Dijkstra monad structure makes these typing rules uniform for all computation types.

Layered indexed effects, in a nutshell. The main contribution of this paper is to generalize these typing rules so that they no longer require all computation types to be Dijkstra monads (§3). Instead, computation types in \( F^* \) are now of the form \( \text{Tot} \ t \) (as before) or are indexed in arbitrary ways as \( M \ t \ i \), where the new rule for \( \text{T-Bind} \) allows the (variable number of) indices \( i \) to be composed in arbitrary, user-controlled ways—all that's required to introduce a layered indexed effect \( M \ t \ i \) is for it to have the following:

1. A representation type, \( M \cdot \text{repr} \ t \ i \)
2. A return, with shape \( a : \text{Type} \rightarrow x : a \rightarrow \text{implicits} \rightarrow M \cdot \text{repr} \ a \ i \)
3. A bind, with shape \( a : \text{Type} \rightarrow b : \text{Type} \rightarrow \text{implicits} \rightarrow M \cdot \text{repr} \ a \ \hat{p} \rightarrow (x : a \rightarrow M \cdot \text{repr} \ b \ \hat{q}) \rightarrow M \cdot \text{repr} \ b \ \hat{i} \)

The \( \text{T-RETURN} \) and \( \text{T-BIND} \) rules for a given \( M \) correspond to applications of the return and bind combinators, with the implicits computed by \( F^* \)'s existing higher-order unification engine. With no restrictions on the indexing structure, layered indexed effects can be used with a diversity of indexed monad-like structures, and programmers benefit from \( F^* \)'s inference and elaboration algorithm for their user-defined effects. One can choose to prove basic laws for user-defined structures (and we often do), but \( F^* \) neither requires nor depends on any particular laws (§3). Particularly when augmented with other features, e.g. rules for subsumption and lifting between computations types, layered indexed effects make programming with a variety of custom indexed effects in \( F^* \) easy.

\(^2\) \( F^* \) also includes a primitive computation type \( \text{Div} \ t \) for potentially non-terminating computations, isolated from its logical core—we discuss this further in §3.2. Additionally, we write \( x : t \rightarrow t' \) as sugar for \( x : t \rightarrow \text{Tot} \ t' \).
Encoding \textsc{pure} as a layered effect. To turn \textsc{pure} into a user-defined computation type, all that’s required is to define the \textsc{pure} type shown earlier, a return and bind for it, and to ask \textsc{f*} to turn it into an effect.

let pure (a:Type) (w:wp a) : Type = p:(a \rightarrow \text{prop}) \rightarrow \text{squash} (wp p) \rightarrow \text{Tot} (v:a[p v])

let return (a:Type) (x:a) : pure (w: \text{return} x) = \lambda _ _ \rightarrow x

let bind (a b:Type) (w1:wp a) (w2:(a \rightarrow wp b)) (f: pure a w1) (g: (x:a \rightarrow pure b (w2 x))): pure b (wbind w1 w2) = \lambda p _ \rightarrow let x = f (\lambda x \rightarrow w2 x p) () in g x p ()

effect \{ \text{pure} \ (\text{a:Type}) (wp a) = \text{pure with} \ \text{return}; \text{bind} \}

The representation we chose for \text{pure} is a form of continuation-passing, where the first argument \text{p} is the postcondition to prove; the second \text{squash} (wp p) is a hypothesis for the corresponding precondition; and the result \text{v:a[p v]} is refined to satisfy the postcondition. In \text{bind}, we first run \text{f} into a value, by calling it with the \((\lambda x \rightarrow w2 x p)\) postcondition, whose precondition according to \text{w1} must hold: it is exactly \text{wbind w1 \text{w2} \text{p}}. Then, we simply call \text{g} with it at postcondition \text{p}. The unit values take the place of the squashed proofs of preconditions, discharged by the SMT solver.

In essence, using \text{f*}’s existing support for dependent types and refinement types, we can give a semantic model for \text{pure}, while the effect definition turns \text{pure} into a computation type \text{pure} so that client programs are agnostic to the choice of representation (several other representations for \text{pure} are also possible) and can simply work with respect to the abstraction provided.

Beyond the bare minimum of providing a return and bind, effect definitions can be augmented with rules for subsumption and for composing conditional computations. For instance, \text{subcomp} below allows weakening a specification (by strengthening its precondition pointwise)—the definition of \text{subcomp} is its own soundness proof. Meanwhile, \text{cond} shows that case analyzing a boolean \text{b} and running either \text{f} or \text{g} produces a \text{pure} computation whose \text{wp}-index reflects the case analysis too—\text{F*} checks that if \text{b} then \text{f} else \text{g} can be given the type claimed by \text{cond} \_ \_ \_ \text{f} \text{g} \text{b}, ensuring its soundness.

let subcomp (a:Type) (w1 w2 : wp a) (u : squash (\forall p . w2 p \implies w1 p)) (f : pure a w1) : pure a w2 = f

let cond (a : Type) (w1 w2 : wp a) (f : pure a w1) (g : pure a w2) (b : bool): Type = pure a (\lambda p \rightarrow (b \implies w1 p) \land ((\neg b) \implies w2 p))
effect \{ \text{pure} \ (\text{a:Type}) (wp a) = \text{pure with} \ \text{return}; \text{bind}; \text{subcomp}; \text{cond} \}

Without these two additional combinators, \text{f*} would do no better than forcing the branches of a case analysis to match precisely (i.e. via provable equality), and there would be no way to, e.g., branch into \text{pure} computations with non-equal indices.

With our layered indexed effect in place, we can use primitive syntax for computations in \text{pure} and leave it to \text{f*}’s inference to compute WPs. As is common in \text{f*}, we also define an alias \text{Pure} so we can write pre- and postconditions instead of WPs when needed, decorating them with the tags \text{requires} and \text{ensures} to help with readability. Below is a small example of a partial map function over a list: the function \text{f} being mapped has an argument-dependent precondition, which is propagated into the precondition of \text{map} via a quantification over the elements of the list.

effect \text{Pure} (a:Type) (pre:prop) (post:a \rightarrow prop) = \text{pure} \ a (\lambda p \rightarrow \text{pre} \land (\forall x . \text{post} x \implies p x))

let rec map #: # # pre (f : (x:a \rightarrow \text{Pure} b (\text{requires} (pre x))) (\text{ensures} (\lambda _ \rightarrow \text{T}))) (l : list a) : \text{Pure} (list b) (\text{requires} (\forall x . x \in l \implies \text{pre} x)) (\text{ensures} (\lambda _ \rightarrow \text{T})) = \text{match} l \text{with} | [] \rightarrow [] | x::xs \rightarrow f x :: \text{map} f xs

Of course, Dijkstra monads may seem overkill for specifying pure computations—one could also have written the type for \text{map} with refinement types alone. However, all other \text{F*} effects are built on top of \text{pure}, and making it a derived notion paves the way to removing it from the core, simplifying
F*'s foundations. Further, being liberated from Dijkstra monads alone, we can develop other effect typing disciplines, as we see will see next.

2.2 Graded Monads and Dijkstra Monads for Algebraic Effects and Handlers

Algebraic effects and handlers are a framework for modeling effects in an extensible, composable, re-interpretable manner, with strong semantic foundations [Plotkin and Power 2003] and several new languages and libraries emerging to support them [Bauer and Pretnar 2015; Brady 2013; Leijen 2017; Lindley et al. 2017; Plotkin and Pretnar 2009]. In this section, we show that algebraic effects and handlers can be encoded generically using dependent types in F*, and exposed to programmers as a layered indexed effect supporting programming in an abstract, high-level style. Further, we show that operation labels can be conveniently tracked as an index, much like in existing effect systems for algebraic effects. Also, we reconcile them with WPs for the particular case of state, proving that functional specifications can be strengthened from the intensional information of its operations, employing a unique combination of graded and Dijkstra monads.

Roadmap. Our starting point is a canonical free monad representation \( \text{tree}_0 \) a of computations with generic actions producing a-typed results. Next, we refine \( \text{tree}_0 \) to a graded monad \( \text{tree} \) (1:list op), where 1 over-approximates the actions in the tree. We then define dependently typed handlers, that allow handling actions in a computation in a custom manner. Packaging \( \text{tree} \) as an indexed effect \( \text{Alg a l wp} \) allows programming with effects and handlers in a direct syntax, with types recording the effectful actions that are yet to be handled; e.g. an \( \text{Alg a []} \) is a pure computation that can be reduced to a result. Further, to verify programs, we give a more refined variant \( \text{AlgWP a l wp} \) introducing an additional WP index, which describes the behavior of the computation with respect to a standard, state-passing implementation. We prove the WP index to be sound, and provide an operation to strengthen the WP of write-free programs, as well as provide some simple examples. Being agnostic to the choice of indexing, layered indexed effects supports equally well effectful programming with graded monads and program verification with Dijkstra monads—both on top of the same algebraic effects framework.

A free monad over generic actions. We begin by defining a type for the operations and giving their input and output types. 3 We include stateful operations (Read and Write) and exceptions (Raise), but other operations can be easily added. 4 Then, we define the type for computations that sequentially compose them according to their arity, i.e. the free monad over them.

3An alternative is to index op with input and output types. This is essentially equivalent, but notationally heavier.

4Our implementation in the supplementary material contains an additional uninterpreted \( \text{Other : int} \to \text{op} \), and never relies on knowing the full set of operations.
so order and multiplicity are irrelevant. This makes tree a graded monad, where the monoid in
question takes the union of sets of operations.

```
val type ops = list op

let rec abides #a (labs:ops) (c : tree0 a) : prop = match c with
  | Return _ -> \top
  | Op a k i l labs = \forall o, abides labs (k o)

val type tree (a:Type) (labs:ops) = c:(tree0 a)[abides labs c]
```

*Packaging the refined free monad as an effect.* Elevating tree to an indexed effect is straightforward,
only requiring to define the basic combinators. With most of the heavy lifting done by F*'s SMT
backend, we do not need to explicitly prove the abides refinements, provided some theorems about
list membership are in scope. We also provide a subsumption rule to grow the labels where needed,
and also reorder and deduplicate them, since they are essentially sets. The branching combinator
joins the labels of each branch. It can be easily verified that these last two satisfy the expected
relation (and F* proves that automatically as well).

```
let return a (x:a) : tree a [] = Return x

let bind a b labs = tree1 (\forall i, a b labs i) (f : a \rightarrow \top b labs) : tree b (\top labs \cup labs) = (\* elided \* )

let subcomp a b labs (f : tree a labs) : Pure (tree a labs) (requires (\top labs \subseteq labs)) = f

let cond (a:Type) (labs : ops) (f:tree a labs) (g:tree a labs) (p:bool) = tree a (\top labs \cup labs)

val effect {Alg (a:Type) (labs:ops) = tree with return; bind; subcomp; cond}
```

*Reflecting operations.* An effect like Alg is an abstract alias for its underlying representation type
tree. Based on an existing feature of F*, itself inspired by Filinski [1999], we can *reify* Alg to its
representation and conversely *reflect* tree as an Alg, revealing the abstraction only when necessary.
Using reflection, algebraic operations can be uniformly turned into computations via geneff below.
For raise, we take an extra step of matching on its (empty) result to make it polymorphic. Simple
programs can already be written in direct style, with automatic inference of the labels—as a matter
of best practice, F* requires documenting the type of top-level definitions, otherwise even those
could be left out and inferred.

```
let geneff (o : op) (i : op_inp o) : Alg (op_out o) [o] = Alg?.reflect (Op o i Return)

let get () : Alg state [Read] = geneff Read ()

let put (s:state) : Alg unit [Write] = geneff Write s

let raise (e:exn) : Alg a [Raise] = match geneff Raise e with (\* empty match \*)

exception Failure of string

let add_st (x y:int) : Alg int [Read;Raise] =
  let s = get () in (\* assuming integer state \*)
  if s < 0 then raise (Failure "error") else x + y + s
```

*Implementing handlers generically.* Programming dependently typed generic handlers for trees
is relatively smooth. The type handler_tree_op o b labs below represents a handler for the single
operation o into a computation of type b and operations in labs. The handler will be called with
the parameter to the operation and its continuation that has already been handled recursively.
A handler_tree labs0 b labs1 is a dependent product of operation handlers over domain labs0;
in other words, one handler for each operation in labs0, all of them coinciding in the type and
operations they handle to. The handle_tree function implements generic handling of a computation
tree c given a value case v and a generic handler h for every operation in c.

```
type handler_tree_op (o:op) (b:Type) (labs:ops) = op_inp o → (op_out o → tree b labs) → tree b labs

let rec handle_tree (c : tree a labs0) (v : a → tree b labs1) (h : handler_tree labs0 b labs1) =
  tree b labs1 = match c with
  | Return x → v x
  | Op act i k → h act i (λ o → handle_tree (k o) v h)

While general, this handling construct over trees is uncomfortable as one needs to manually bind and return computations, which may become unwieldy as trees nest, e.g., the usual handler for Read/Write into the state monad involves a (tree (state → tree (a & state) labs) labs). It is much more convenient to define an effectful handling construct directly usable in Alg. To do so, we can simply lift handle_tree into Alg via reflection (and reification in negative positions). Handling via handle_with can be done in direct style, with full support for label inference and automatic elaboration into monadic form.

let handle_with (#a #b #labs0 #labs1)
  (f : unit → Alg a labs0) (v : a → Alg b labs1) (h : handler_tree labs0 b labs1) : Alg b labs1
  = (elided, essentially a wrapper of handle_tree, translating h into a handler_tree, etc. *)

We show below the handler of Raise into the option monad, and that of Read/Write into the state monad, both of which are written in direct style and leave other operations unhandled. To only handle part of the operations within a computation, we add a default case to h where needed. This can be done generically via defh below: it defines a handler that leaves every operation unhandled by executing it and calling the continuation. Accordingly, its type states that it does not modify the label set. Thanks to unification, all of its arguments are usually inferred, even the operation; in our implementation we make also o implicit and elide the underscore argument. However, catchST does requires a small annotation: F* needs the information about the nesting of effectful computations, and about the set of internal labels. Barring that, everything else (lifting, binding, weakening, etc) is handled seamlessly by the effect system. The applicative syntax in the handler is similar to that of other languages specifically designed for algebraic effects [Bauer and Pretnar 2015; Leijen 2014].

let defh #b #labs : handler labs b labs = λ o i k → k (geneff o i) (* essentially rebuilding an Op node *)

let catchE #a #labs (f : unit → Alg a (Raise::labs)) : Alg (option a) labs =
  handle_with f Some (function Raise → λ i k → None
  | _ → defh _)

let catchST #a #labs (f : unit → Alg a (Read::Write::labs)) (s0:state) : Alg (a & state) labs =
  handle_with #_ #state → Alg _ labs`) #_ labs
  f (λ x s → (x, s)) (function Read → λ k s → k s s
  | Write → λ s k _ → k () s
  | _ → defh _) s0

Running programs. To run computations, we can handle all their operations and extract the final result. As usual, the choice of how to interpret operations is given by the handlers. In the example below we show how programs combining state and exceptions can be interpreted into the two most common monadic layouts, according to the stacking of the relevant monad transformers. The order of the labels of f is not important at all, even for inference: in both examples unification will work inwards from the fact that run takes a label-free computation and directly infer a list of labels for f. It will then use subcomp to check that it abides by them, allowing re-orderings.
We also prove the WP calculus sound by interpreting a weaker than its annotated WP (allowing underspecification). This construction is due to Maillard et al. Let read_wp

let run_STExn #a (f: unit → Alg a []) : Tot a = match reify (f ()) with | Return x → x
let run_STExn #a (f: unit → Alg a [Write; Raise; Read]) (s₀: state) : option (a & state) = run (λ () → catchE (λ () → catchST f s₀))

AlgWP: A graded Dijkstra monad for stateful Alg programs. While Alg above bounds the operations each computation may invoke, there is no way to specify their order, or any property about the final value and state. To do so, we can bring back the idea of using a WP calculus to the algebraic setting by adding a new index to the effect. We limit ourselves to stateful operations and use WPs to specify the behavior according to the state monad: without fixing an interpretation, there is not much to be said in terms of verification unless equations are added. That is the topic of future work in the area, but we believe the effectful representation would not be heavily affected by it.

Our new effect will refine tree even further. The main idea is to begin with a function that takes a computation tree and computes a "default" stateful WP from it. Then, we take the representation tree_wp a l wp to be the computation trees with operations in l whose default WP is pointwise weaker than its annotated WP (allowing underspecification). This construction is due to Maillard et al. [2019] but our setting is more general due to the additional grading, not supported in Dijkstra monads. We begin by defining a predicate transformer monad for stateful programs.

1:10 Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy

let rec interp_as_wp #a (t: tree a [Read; Write]): st_wp a = match t with
  | Return x → return_wp x
  | Op Read _ k → bind_wp (read_wp (λ s → interp_as_wp (k s)))
  | Op Write s k → bind_wp (write_wp s (λ () → interp_as_wp (k ())))

type rwops = l:ops[{l ⊆ [Read; Write]}
let treewp (a: Type) (l:rwops) (w: st_wp a) = t:(tree a l)({ w ∈ interp_as_wp t })
effect { AlgWP (a: Type) (l:rwops) (w: st_wp a) = treewp with ...}
let get () : AlgWP state [Read] read_wp = AlgWP?.reflect (Op Read () Return)
let put (s:state): AlgWP unit [Write] (write_wp s) = AlgWP?.reflect (Op Write s Return)
let soundness #a #l #wp (t: unit → AlgWP a l wp): s₀:state → PURE (a & state) (wp s₀) = ...

Using this graded Dijkstra monad, we can verify functional correctness properties, which a graded monad alone cannot capture. For instance, when the state is instantiated to a heap (mapping locations to values), we can prove that the program below correctly swaps two references in the heap—AlgPP is simply a pre-/postcondition alias to AlgWP, in the style of Pure in §2.1.
let swap (l₁ l₂ : loc) : AlgPP unit [Write; Read] (requires λₗ → l₁ ≠ l₂)
(ensures λh₀ λₗ h₁ → h₀\{l₁;l₂} ∧ h₁,lₗ = h₀,l₁ ∧ h₁,l₁ = h₀,l₁)
= let r = !l₂ in l₂ := !l₁; l₁ := r

Of course, this is a simple example, and verification of stateful programs is a discipline in itself, but it illustrates how different layers and styles of verification can interact within a single language.

More interestingly, the static information in the label index can be exploited by the WP. The quotient function below strengthens the postcondition of a write-free AlgWP program into additionally ensuring that the state does not change. Operationally, quotient just runs f(), so it can be seen as a proof that f does not change the state. On the other hand, ignore_writes below handles all Write operations in f by ignoring them. In this case the postcondition becomes possibly weaker: it says nothing about the result, but still ensures that the state does not change.

val quotient #a #pre #post (f : unit → AlgPP a [Read] pre post)
: AlgPP a [Read] pre (λh₀ x h₁ → post h₀ x h₁ ∧ h₀ = h₁)
val ignore_writes #a (#labs:rwops[Write ≡ labs]) #wp (f : unit → AlgWP a [Write::labs] wp)
: AlgWP a labs pre (λh₀ λₗ h₁ → h₀ = h₁)

We have barely scratched the surface of algebraic effects and handlers in F*, but we see it as a very promising area for further work. That these disciplines can be prototyped so straightforwardly is a testament to the usability of layered indexed effects.

2.3 A Graded Monad for Stateful Information Flow Control

In this section, we present an indexed effect to track information flows [Denning 1976] in higher-order, stateful programs. Our general approach is to adopt the techniques of monadic information flow control (IFC), in the tradition of Abadi et al.’s (1999) dependency core calculus and others [Crary et al. 2005; Devriese and Piessens 2011; Hicks et al. 2014; Russo et al. 2008], though expressed as a layered indexed effect based on a graded monad. We enhance prior approaches in three notable ways. First, rather than tracking flows among a given lattice of information flow control labels, our approach tracks flows between arbitrarily fine-grained sets of memory references—working in a dependently typing setting, dynamic IFC labels [Zheng and Myers 2007] are very natural. Second, we develop a novel graded monad in which to separately track read and write effects as well as a flows relation, which records dependences among the reads and writes. Finally, unlike prior approaches which have relied on a separate metatheory for security, we prove a noninterference theorem entirely within the system. Yet, the semantic model does not pollute client reasoning: by packaging our graded monad as an effect layer, client F* programs can be verified for noninterference by analysis of the flows relation, as one might expect in a custom information-flow analysis, hiding all details of the underlying semantic model.

A two-step plan. We start by presenting a relational model of read and write effects, in the spirit of Benton et al. [2006]. We develop a graded monad rwst a r w, describing the type of an a-returning computation that may read the memory locations r and write the locations w. This model is already sufficient for many purposes (e.g., to justify the soundness of various program optimizations), as well as to track information flow. Indeed, one can prove the information flows in an rwst a r w program are contained in the flows between the read set r and the write set w. However, as we will see, this model is also overly conservative. Addressing this imprecision, we present the main construction of this section, an indexed effect IFC a r w fflows, a refinement of rwst which additionally tracks a subset of the reads that can influence a subset of the writes.

Tracking read and write effects. We consider computations in a state monad st a = store → a & store where store = Map.t loc int, maps abstract memory locations loc to integers—we do not
model dynamic allocation nor typed references, as they are orthogonal. Our first graded monad, \( \text{rwst} \) is the refinement of \( \text{st} \) shown below.

\[
\text{let } \text{rwst} \ (a : \text{Type}) \ (r : \text{label}) \ (w : \text{label}) = f : \text{st a} \{ \text{reads\_bounded} \ f \ r \land \text{writes\_bounded} \ f \ w \}
\]

\[
\text{where}
\]

\[
\text{let } \text{label} = \text{set loc}
\]

\[
\text{let } \text{writes\_bounded} \ f \ w = \forall l . l \not\in w \implies (\forall s . \text{sel s} (\text{snd f s}) l = \text{sel s l})\ (\text{\ast locs not in w do not change \ast})
\]

\[
\text{let } \text{reads\_bounded} \ f r = \forall l . l \not\in r \implies (\forall s . \text{sel s l})\ (\text{\ast runs the computation twice \ast})
\]

\[
\text{let } \text{write\_read} \ (a : \text{Type}) \ (r : \text{label}) \ (w : \text{label}) = \text{rwst} \ a \ r \ w
\]

Constraining the write effects in \( \text{rwst} \) is relatively straightforward: \( \text{writes\_bounded} \ f \ w \) simply states that the contents of all locations \( l \) not in the write set \( w \) are unchanged when running \( f \) in any state. Specifying read effects is bit more subtle—\( \text{reads\_bounded} \ f \ r \) captures the essence of noninterference. It states, roughly, that for any location \( l \) not in the read set \( r \), the result and final state of the computation do not depend on the contents of \( l \). That is, when running \( f \) in any pair of states \( s_0 \) and \( s_0' \) that differ only on the contents of \( l \), the final results \( x \) and \( x' \) are identical; the final states \( s_1 \) and \( s_1' \) are identical on all locations different from \( l \); and \( l \) itself is either written (in which case its contents are the same in both final states), or it is unchanged.

It is relatively straightforward to define the following monadic combinators and actions for \( \text{rwst} \), and to package it as a layered effect.

\[
\text{let } \text{return} \ #a \ (x : a) : \text{rwst a} \ {} {} = \lambda s \to x, s
\]

\[
\text{let } \text{bind} \ #a \ #b \ (f : \text{rwst} a \ r_0 \ w_0) \ (g : a \to \text{rwst} b \ r_1 \ w_1) : \text{rwst} b \ (r_0 \cup r_1) \ (w_0 \cup w_1) = \ldots
\]

\[
\text{\textbf{effect}} \ \text{ RWST (a : Type) (r : label) (w : label) = \text{rwst with return; bind}}
\]

\[
\text{let } \text{read\_write} \ (l : \text{loc}) : \text{RWST int} \ {l} = \text{RWST?} \ . \text{reflect} \ (\lambda s \to \text{sel s l, s})
\]

\[
\text{let } \text{write\_read} \ (l_0 : \text{loc}) : \text{RWST unit} \ {l_0} \ {l_1} = \text{write l_1 (read l_0)}
\]

\[
\text{However, the abstraction of \( \text{rwst} \) is fairly imprecise inasmuch as the indexing structure cannot distinguish \text{read\_write} above from \text{write\_read} below, even though the latter clearly does not leak the contents of } l_0 \text{ to } l_1.
\]

\[
\text{let } \text{write\_read} \ (l_0 : \text{loc}) : \text{RWST int} \ {l_0} \ {l_1} = \text{write l_1 0; read l_0}
\]

\[
\text{\textit{Refining reads and writes with a flows relation.}} \text{ Our second step involves refining \( \text{rwst} \) to the ifc graded monad shown below, which includes an additional index } fs : \text{flows} \text{ describing the information flows in a computation as a list of edges, where } (\text{from, to}) \in \text{fs} \text{ indicates a potential information flow from the set of memory locations in the "source label" from to the set of locations in the "sink label" to. The } \text{respects} \text{ relation states that if } (\text{from, to}) \text{ is not in the } \text{flows} \text{ relation, then running } f \text{ twice in states that differ only on the contents of from produces states that agree on the contents of to.}
\]

\[
\text{let } \text{ifc} \ a \ (r : \text{label}) \ (w : \text{label}) \ (fs : \text{flows}) = f : (\text{rwst} a \ r \ w) \{ f \, \text{respects} \, fs \}
\]

\[
\text{where}
\]

---

let flow = label & label   let flows = list flow
let has_flow from to f = from ∈ fst f ∧ to ∈ snd f
let has_flow from to fs = ∃ f. fs ∈ fs ∧ has_flow from to f
let no_leak (f:st a) (from to:loc) = ∀ s₀ k. sel (snd (f s₀)) to == sel (snd (f (upd s₀ from k))) to
let respects (f:st a) (fs:flows) = ∀ from to. ¬(has_flow from to fs) ∧ from#to → no_leak f from to

With this refinement, we can define a more precise graded monad structure, as shown below.

let return (x:a) : ifc a [] [] = λ s → x, s
let bind (f:ifc a r₀ w₀ f₀) (g:a → ifc b r₁ w₁ f₁)
 : ifc b (r₀ ∪ r₁) (w₀ ∪ w₁) (f₀ @ add_src r₀ ([], w₁)::fs₁) = ...
where add_src r fs = List.map (λ (src, sink) → (src ∪ r, sink)) fs

The proof of the correctness of bind is non-trivial and requires about 100 lines of auxiliary lemmas. However, once sealed into the effect layer IFC (as shown below), we can write and verify source programs reasoning only about labels and flow relations—the proof relating labels and flow relations to the underlying semantic model is done once and for all.

Subsumption. Further, we can enhance our model with a subsumption relation based on a partial order on labels and flow relations.

let included_in (fs₀ fs₁ : flows) = 
∀ fs₀, fs ∈ fs₀ == (∀ from to has_flow from to f₀ == (∃ f₁. f₁ ∈ fs₁ ∧ has_flow from to f₁))
let ifc_sub (f: ifc a r₀ w₀ f₀ | r₀ ⊆ r₁ ∧ w₀ ⊆ w₁ ∧ f₀ 'included_in' fs₁) : ifc a r₁ w₁ f₁ = f
effect { IFC (a:Type) (r:label) (w:label) (fs:flows) = ifc with return; bind; ifc_sub }

With these structures in place, our sample programs which could not be distinguished by RWST, are correctly distinguished by the flows relation of IFC, with proofs entirely automated by reasoning using the abstraction of labels and the flows relation only.

let read_write (l₀ l₁:loc): IFC unit {l₀} {l₁} [[l₀], {l₁}] = write l₁ (read l₀)
let write_read (l₀ l₁:loc): IFC int {l₀} {l₁} [] = write l₁ 0; read l₀

Of course, label-based IFC remains inherently imprecise. For example, the following typing suggests a flow from l₀ to l₁, even though the result of read l₀ is discarded.

let noop (l₀ l₁:loc): IFC unit {l₀, l₁} [{l₀, l₁}] = let _ = read l₀ in write l₁ (read l₁)

In principle, one could rely on a semantic proof in F* of noop to show that it does not actually leak any information and refine its type—we leave mingling label-based noninterference with semantic proofs to future work.

Having computed the flows in a computation, one can simply check whether the flows exhibited are permissible or not. For example, one could build a traditional lattice of security labels as sets classifying memory locations and check the following:

val high : label   let low = all\{high\}   let lref = l:loc{l ∈ low}   let href = l:loc{l ∈ high}
let read_write (l:lref) (h:href): IFC unit low high [low, high] = write h (read l)

Is ifc really a graded monad? Some of the prevailing wisdom is that static enforcement of stateful information-flow control does not fit into the framework of graded monads. Indeed, both Tate [2013] and Hicks et al. [2014] propose more unusual indexing structures based on effectors or polymonads for IFC. However, our ifc construction shows that it can indeed be done as a graded monad, with a level of precision greater than Hicks et al.’s polymonadic information flow control.

To see that ifc is a graded monad, we need to prove that its indexing structure is a monoid with return indexed by the unit of the monoid and bind’s corresponding to composition of indices in the
monic. That \texttt{rst} is a graded monad should be obvious, since its indices are merely sets composed with \texttt{∪}. What remains is a monoid structure for the \texttt{flows} relation, but composing the flows indices depends on the read and write sets. As such, one can combine the three indices of \texttt{ifc} into a single \texttt{ix} structure, and define a candidate monoid with unit \texttt{e} and composition \texttt{⊕}.

\begin{verbatim}
let ix = label & label & flows
let (e) = {}, {}, []
let (θ) (f₀, w₀, f₁) (r₁, w₁, f₂) = r₀ ∪ r₁, w₀ ∪ w₁, f₀ @ add_src r₀ (({}, w₁):f₁)
\end{verbatim}

This structure forms a monoid under a suitable equivalence relation for \texttt{ix}, which corresponds to equivalence on the label sets and equivalence of \texttt{flows}, as shown below. Pleasantly, \texttt{flows_equiv} also satisfies an intuitive algebraic structure that distributes with \texttt{∪} on labels.

\begin{verbatim}
let flows_equiv : erel flows = λf₀ f₁ → f₀ ´ included_in ´ f₁ ∧ f₁ ´ included_in ´ f₀
let included_in_union_distr (a b c : label)
  : Lemma (flows_equiv [(a, b ∪ c)] [(a, b); (a, c)] ∧ flows_equiv [(a ∪ b, c)] [(a, c); (b, c)]) = ()
\end{verbatim}

### 3 FORMALIZATION AND IMPLEMENTATION OF LAYERED INDEXED EFFECTS

Ahman et al. [2017] present EMF*, a pure type system extended with refinement types and Dijkstra monads, where effectful terms are represented in explicitly monadic form. To model layered indexed effects, we have no need for EMF*’s effectful features nor its support for Dijkstra monads, as it can be encoded. We refer to the effectless fragment of EMF*, including total functions and refinement types, as TotalF* and use it as the target language for elaborating LIF*, a language with layered indexed effects intended as a model of F*’s surface language. The main result of this section is that elaboration is type-preserving, implying that typing derivations in LIF* can be soundly interpreted in TotalF*. We also discuss several implementation aspects of layered indexed effects in the F* typechecker.

#### 3.1 LIF*: Elaborating User-defined Effects

LIF* models the surface language of F* with support for user-defined layered indexed effects. Figure 1 shows the syntax and selected typing rules.

\begin{verbatim}
LIF* adds effectful constructs to TotalF*. These include the monadic computation types \( F t \overline{t} \) (where \( F \) is the effect label, \( t \) is the return type, and \( \overline{t} \) are the effect indices) in \( C \), monadic let-bindings, and explicit \texttt{reify} and reflect coercions to go back-and-forth between computation types and their underlying representations. An effect definition \( D \) defines a layered indexed effect \( F \) with indices types \( x : t \) and combinators \( e_{repr}, e_{return}, e_{bind}, \) and \( e_{subcomp} \), while \( lift^{M_{i}}_{M_{j}} \) defines a combinator to lift \( M_{i} \) computations to \( M_{j} \) (\( M \) ranges over \( \text{Tot} \) and \( F \)). LIF* inherits from TotalF* proof-irrelevant refinement types, dependent functions, standard dependent pattern matching with case (with explicitly annotated return computation type \( C \)), and a non-cumulative predicative universe hierarchy (\( Type_{u} \)).
\end{verbatim}

The ascription form \( e:C \) ascribes \( e \) with computation type \( C \). The implicit monadic structure in the terms is elaborated into explicit binds and lifts by the typing judgment.

**Type-directed elaboration.** The main typechecking judgment in LIF* has the form \( Δ ⊢ e : C \rightsquigarrow e' \) stating that under a typing context \( Δ \), expression \( e \) has computation type \( C \) and elaborates to expression \( e' \) in TotalF*. Before we explain the selected typing rules shown in Figure 1, we discuss typechecking the effect definitions and lifts.

While LIF* does not impose any constraints on the layering or indexing structure, the types of the combinators in an effect definition \( D \) are constrained to have specific shapes. For example, \( e_{bind} \)
We use \( \text{lift} \) when it is clear from the context. Typing rules they are picked non-deterministically in the declarative style while, as we discuss later, \( \text{bind} \) desugars to function application. Finally we write \( F.\cdot \) for an effect \( F \) as typechecked to have a monadic bind shape as (abbreviating \( t_1 : Type \rightarrow t_2 : Type \) as \( t_1, t_2 : Type \)):

\[
S; \cdot \vdash e_{\text{bind}} : t_1, t_2 : Type \rightarrow \overline{\overline{t}} \rightarrow e_{\text{repr}} t_1 \overline{\overrightarrow{t}} \rightarrow (x : t_1 \rightarrow e_{\text{repr}} t_2 \overline{\overrightarrow{t}}) \rightarrow e_{\text{repr}} t_2 \overline{\overrightarrow{t}} \rightarrow \_\_\_\_.
\]

Here \( \overline{\overline{t}} \) are arbitrary binders that may appear in the indices \( \overline{\overrightarrow{t}} \), \( \overline{e} \), and \( \overline{\overrightarrow{t}} \). Similarly, a \( F.\cdot \) is typechecked as a coercion that takes \( F \) computations to \( F' \):

\[
S; \cdot \vdash \text{lift}_{F.\cdot} : t : Type \rightarrow \overline{\overline{t}} \rightarrow F.\cdot e_{\text{repr}} t \overline{\overrightarrow{t}} \rightarrow F'.\cdot e_{\text{repr}} t \overline{\overrightarrow{t}} \rightarrow \_\_\_\_
\]

Every user-defined effect in LIF* gets an automatic lift from \( \text{Tot} \): \( \text{lift}_{F.\cdot} = F.e_{\text{return}}. \)

We now explain the typing rules from Figure 1. We adopt some notational conventions for brevity. We use \( C^t \) to project the return type component from a \( C \). \( \acute{C} \) is the underlying representation of \( C \) defined as \( t \) when \( C = \text{Tot} \) and \( F.e_{\text{repr}} t \overline{\overrightarrow{t}} \) when \( C = F t \overline{\overrightarrow{t}} \). We also elide the type \( t \) from \( \lambda x : t.\cdot e \) when it is clear from the context. \( M.\text{bind} \) projects the bind combinator for the effect \( M \) (for \( M = \text{Tot} \), bind desugars to function application). Finally we write \( \text{lift}_{C^t} \) to mean \( \text{lift}_{M^t} \) where \( M^t \) is the effect label of \( C^t \). For applications of the effect combinators, we elide the non-repr arguments; in the full typing rules they are picked non-deterministically in the declarative style while, as we discuss later, our implementation infers them using higher-order unification.

---

**Fig. 1.** Syntax of LIF* and selected typing judgments (primed symbols are \( \text{Tot} \) syntax)
Rule T-Var is the standard variable typing rule with identity compilation to TotalF*. Rule T-Let is the let-binding rule. Whereas EMF* had explicit monadic binds and lifts in the syntax, in LIF*, monadic elaboration is type-directed and more accurately describes our implementation. The rule first typechecks $e_1 : C_1$ and $e_2 : C_2$. Since $C_1$ and $C_2$ could have different effect labels, the rule lifts them to a common effect $M$, and binds the resulting $M$ computations. The rule is reminiscent of Swamy et al.’s (2011) and Hicks et al.’s (2014) monadic elaboration rules, though both their calculi are non-dependent. Concretely, the rule introduces two fresh variables $f : \hat{C}_1$ and $g : x:C'_1 \rightarrow \hat{C}_2$, applies the lift combinators to $f$ and $g$, and then applies the resulting computations to $M$. The let-binding is assigned the computation type $C$ and the compiled TotalF* term is $e'$ with $e_1'$ and $e_2'$ substituted for $f$ and $g$.

Rule T-Case is similar. It first typechecks the scrutinee $e$ and the two branches $e_1$ and $e_2$ under appropriate assumptions. The rule then lifts the two branches to an effect $M$ by applying the lift combinators to fresh variables $f_1$ and $f_2$, and constructs the final TotalF* term with appropriate substitutions as in T-Let. T-Reify and T-Reflect move back-and-forth between a computation type and its representation. Interestingly, the elaboration of reify $e$ (resp. M.reflect $e$) is just the elaboration of $e$; reify and reflect are just identity coercions in LIF* with no counterpart necessary in TotalF*. In contrast, Filinski [1999] uses monadic reflection to structure the compilation of monadic computations using state and continuations—we leave exploring this possibility to the future, which may allow for more efficient compilation of user-defined effects.

We do not have the standard declarative subsumption rule for expression typing; instead we rely on the rule T-As to explicitly trigger the subtyping of the computation types. With the placement of lift coercions also directed by the syntax (rules T-Let and T-Case), this makes our typing judgment purely syntax-directed. We discuss the coherence of multiple lifts and least upper bound of effects in §3.2.

Rule C-M typechecks a computation type $C$ by typechecking $\hat{C}$. The computation type subtyping rule SC-M delegates to subtyping on the underlying representations. Rule S-Conv shows the subtyping rule based on type conversion using reduction; it compiles the two types and reduces them in TotalF*. Similarly, the refinement subtyping rule (elided for space reasons) appeals to the logical validity judgment in TotalF* after compiling the two types. We also elide the standard arrow subtyping, reflexivity, and transitivity rules.

Our main theorem states that the LIF* translation to TotalF* is well-typed.

**Theorem 3.1.** If $S; \Gamma \vdash e : C \rightsquigarrow e'$, then $S; \Gamma \vdash C : Type \rightsquigarrow t'$ and $[[ \Gamma ]]_S e' : t'$.

Here, $[[ \Gamma ]]_S$ is the pointwise translation of the typing environment, and $[[ \Gamma ]]_S e' : t'$ is the typing judgment in TotalF*. Using the theorem, a typing derivation in LIF* can be soundly interpreted in TotalF*. Ahman et al. [2017] prove EMF* normalizing, type-preserving, and a consistency property for its refinement logic—these results also apply to its TotalF* fragment. Still, TotalF* is not yet sufficient to model all of the Γ* implementation, lacking a few important features, notably, partial correctness for Swamy et al.’s (2016) divergence effect, universe polymorphism, type conversion using provable equality, and inductive types. We plan to study these enhancements in the future, grateful to no longer have to consider their interaction with Dijkstra monads.

The proof of the theorem is by mutual induction on the typing derivation with the following lemmas:

**Lemma 3.2 (Commutation of Subtyping).**

(a) If $S; \Gamma \vdash t <: t_1$ and $S; \Gamma \vdash t : Type \rightsquigarrow t'$ then $S; \Gamma \vdash t_1 : Type \rightsquigarrow t'_1$ and $[[ \Gamma ]]_S t' <: t'_1$.

(b) If $S; \Gamma \vdash C <: C_1$ and $S; \Gamma \vdash C : Type \rightsquigarrow t'$ then $S; \Gamma \vdash C_1 : Type \rightsquigarrow t'_1$ and $[[ \Gamma ]]_S t' <: t'_1$.

The essence of effect abstraction: Admissibility of using $\epsilon_{\text{subcomp}}$ for subtyping. The reader may have noticed that the rule SC-M breaks the effect abstraction by peeking into the effect representation for checking subtyping. However, this is not necessary: we show the admissibility of checking subtyping using the $\epsilon_{\text{subcomp}}$ effect combinator.

The $\epsilon_{\text{subcomp}}$ combinator for a user-defined effect $F$ is typechecked as follows:

$$S; \vdash \epsilon_{\text{subcomp}} : t : \text{Type} \rightarrow \overline{x} : \overline{t} \rightarrow f : \epsilon_{\text{repr}} t \overline{e}_1 \rightarrow \epsilon_{\text{repr}} t \overline{e}_1 \leadsto_{\gamma}$$

and

$$\epsilon_{\text{subcomp}} = \lambda t \overline{x} f. f$$

where $t_1$ is a refinement formula. The intuition is that the combinator is a coercion that can be used to coerce a computation from $F t \overline{e}$ to $F t \overline{e}_1$, provided that the refinement formula is valid. The implementation of $\epsilon_{\text{subcomp}}$ is required to be an identity function. This is because our subtyping relation is currently non-coercive. To prove $F t \overline{e} : F t \overline{e}_1$, we check that:

$$S; \Gamma; f : F.\epsilon_{\text{repr}} t \overline{e} \vdash F.\epsilon_{\text{subcomp}} f : F.\epsilon_{\text{repr}} t \overline{e}_1 \leadsto_{\gamma}$$

Behind the scenes, this typechecking judgment proves the refinement formula in the type of the $\epsilon_{\text{subcomp}}$ combinator. The following lemma establishes the soundness of $\epsilon_{\text{subcomp}}$:

**Lemma 3.3 (Soundness of $\epsilon_{\text{subcomp}}$).** If $\Delta \vdash F t \overline{e} : \text{Type} \leadsto_{\gamma} \Delta \vdash F t \overline{e}_1 : \text{Type} \leadsto_{\gamma}$, and $\Delta, f : F.\epsilon_{\text{repr}} t \overline{e} \vdash F.\epsilon_{\text{subcomp}} f : F.\epsilon_{\text{repr}} t \overline{e}_1 \leadsto_{\gamma}$, then $\Delta \vdash F t \overline{e} : F t \overline{e}_1$.

The proof of the lemma unfolds the applications of $\epsilon_{\text{subcomp}}$ to prove the subtyping of the representations, after which an application of SC-M gives us the conclusion. In our implementation, we use the $\epsilon_{\text{subcomp}}$ combinator rather than implementing the SC-M rule as is, ensuring that effect abstractions are preserved.

### 3.2 Implementation of Layered Indexed Effects

We have implemented layered indexed effects in the F* typechecker and all the examples presented in the paper are supported by our implementation. Below we discuss some implementation aspects.

*Coherence of lifts and effect upper bounds.* When adding a lift $M_1$ to the signature, our implementation computes all the new lift edges that it induces via transitive closure. For example, if lift $M_0$ and lift $M_1$ already exist, this new lift induces a lift $M_1$ via composition. For all such new edges, if the effects involved already have an edge between them, F* ignores the new edge and emits a warning as such. Further, F* also ensures that for all effects $M$ and $M_1$, either they cannot be composed or they have a unique least upper bound. This ensures that the final effect $M$ is unique in the rules T-LET and T-CASE. Finally, F* ensures that there are no cycles in the effects ordering.

*Effect combinator for composing branches of a conditional.* While in LIF we have formalized a dependent pattern matching case, our implementation allows for specifying an optional custom effect combinator for combinimg branches. The shape of the combinator is as follows:

$$S; \vdash \epsilon_{\text{cond}} : t : \text{Type} \rightarrow \overline{x} : t \rightarrow f : \epsilon_{\text{repr}} t \overline{e}_{\text{then}} \rightarrow g : \epsilon_{\text{repr}} t \overline{e}_{\text{else}} \rightarrow b : \text{bool} \rightarrow \text{Type} \leadsto_{\gamma}$$

and

$$\epsilon_{\text{cond}} = \lambda t \overline{x} f g b. \epsilon_{\text{repr}} t \overline{e}_{\text{composed}}$$

F* ensures the soundness of the combinator by checking that under the assumption $b$, the type of $f$ is a subtype – as per the effect subcomp combinator – of the composed type, similarly for $g$ under the corresponding assumption not $b$. When the combinator is omitted, F* chooses a default one that forces the computation type indices for the branches to be provably equal.

*Inferring effect indices using higher-order unification.* Our implementation relies on the higher-order unifier of F* to infer effect indices and arguments of the effect combinators. For example, suppose we have a computation type $F t_1 \overline{e}_1$ and we want to apply the lift $F$ combinator, where:
\[ S; \cdot \vdash \text{lift}_{F'.e} : t : \text{Type} \rightarrow \overline{x}; t \rightarrow F.\text{e}_{\text{repr}} t \overline{e} \rightarrow F'.\text{e}_{\text{repr}} t \overline{e} \sim \text{_} \]

To apply this combinator, we create fresh unification variables for the binders \( t \) and \( \overline{x} \), and substitute them with the unification variables in \( F.\text{e}_{\text{repr}} t \overline{e} \) and \( F'.\text{e}_{\text{repr}} t \overline{e} \), without unfolding the \( \text{e}_{\text{repr}} \). We then unify \( t_1 \) with the unification variable for \( t \), \( t_2 \) with substituted \( \overline{e}_1 \), and return (substituted) \( F' t \overline{e} \) as the lifted computation type. This allows us to compute instantiations of the combinators without reifying \( F t_1 \overline{e}_1 \) or reflecting the result type of lift. We follow this recipe for all the effect combinators.

**Support for divergence.** Our implementation also supports the existing \( \text{Div} \) effect in \( F^* \) for classifying divergent computations. To ensure consistency, the logical core of \( F^* \) is restricted to the pure fragment, separated from \( \text{Div} \) using the effect type system. When defining layered indexed effects, the \( \text{repr} \) types may encapsulate \( \text{Div} \) computations. A layered indexed effect \( M \) may optionally be marked divergent. When so, the semantic termination checker of \( F^* \) is disabled for \( M \) computations. However, reification of such effects results in a \( \text{Div} \) computation to capture the fact that this computation may diverge.

**Polymonads.** Polymonads are closely related to layered indexed effects. Hicks et al. [2014] define a polymonad as a collection of unary type constructors \( M \) and a family of bind-like operators with signature \( M_j : a \rightarrow (a \rightarrow M_j b) \rightarrow M_k b \), together with a distinguished element of \( M \) called \( \text{Id} \), corresponding to an identity monad. In our dependently typed setting, we consider all the \( M_i \) to be instances of the same indexed effect constructor, with the indices varying arbitrarily in \( \text{bind} \), and where \( \text{Id} \) corresponds to \( \text{Tot} \). Sometimes, however, it can be convenient to also allow the effect constructors to vary in \( \text{bind} \) (see §5 for an example). As such, our implementation also supports writing \( \text{polymonadic}_{\text{bind}} (F_1, F_2) \triangleright e = e \), where \( e \) is typechecked and applied similarly to the single-monadic bind shown earlier, i.e.,

\[ S; \cdot \vdash e : t_1 t_2 : \text{Type} \rightarrow \overline{x}; t \rightarrow F_1.\text{e}_{\text{repr}} t_1 \overline{e}_f \rightarrow (x : t_1 \rightarrow F_2.\text{e}_{\text{repr}} t_2 \overline{e}_g) \rightarrow F.\text{e}_{\text{repr}} t_2 \overline{e} \sim \text{_} \]

Our implementation also allows writing \( \text{polymonadic}_{\text{subcomp}} F_1 <: F_2 = e \), a construct that allows lifting and subsuming \( F_1 \) to \( F_2 \) in a single step. In a way, polymonadic binds and subcomps allow programmers to construct a polymonad openly and incrementally, with parallels to, e.g., the instances of a typeclass. Both these constructs offer programmers finer control when combining and relating computations in multiple effects, while the typechecker makes sure that the effect ordering is still coherent.

## 4  CASE STUDY: MESSAGE FORMATTING FOR TLS

Layered indexed effects are not just for defining new effect typing disciplines—effect layers stacked upon existing effects can make client programs and proofs more abstract, without any additional runtime overhead. We demonstrate this at work by layering an effect over EverParse [Ramananandro et al. 2019], an existing library in \( F^* \) for verified low-level binary message parsing and formatting.

**Background: EverParse and Low*.** EverParse is a parser generator for low-level binary message formats, built upon a verified library of monadic parsing and formatting combinators. It produces parsers and formatters verified for memory-safety (no buffer overruns, etc.) and functional correctness (the parser is an inverse of the formatter). EverParse is programmed in Low*, a DSL in \( F^* \) for C-like programming [Protzenko et al. 2017]. Low*’s central construct is the \( \text{Stack} \) effect which models programming with mutable locations on the stack and heap, with explicit memory layout and lifetimes. Its signature, given below, shows a Hoare monad.

\[ \text{effect Stack} (a : \text{Type}) \triangleright (\text{pre}_{\text{mem}} \rightarrow \text{prop}) (\text{post}_{\text{mem}} \rightarrow a \rightarrow \text{mem} \rightarrow \text{prop}) \]

Programs in \( \text{Stack} \) may only allocate on the stack, while reading and writing both the stack and the heap, with pre- and postconditions referring to \( \text{mem} \), a region-based memory encapsulating both
stack and heap. Low* provides fine-grained control for general-purpose low-level programming, at the expense of low-level proof obligations related to spatial and temporal memory safety and framing. When programming specialized code such as binary message formatters, layered indexed effects offer a way to build domain-specific abstractions, liberating programs from memory safety proofs and many details of low-level message formats.

The problem: Existing code mired in low-level details. Consider, for instance, formatting a struct of two 32-bit integer fields into a mutable array of bytes, a buffer U8.t in Low* parlance.

type example = {left:U32.t; right:U32.t}

EverParse generates a lemma stating that if the output buffer contains two binary representations of integers back to back, then it contains a valid binary representation of an example struct value:

val example_intro mem (output: buffer U8.t) (offset_from: U32.t): Lemma
  (requires valid_from parse_u32 mem output offset_from ∧
   valid_from parse_u32 mem output (offset_to parse_u32 mem output offset_from))
  (ensures valid_from parse_example mem output offset_from)

To format a value of this type, one must write code like this:

let write_example (output: buffer U8.t) (len: U32.t) (left right: U32.t) : Stack bool
  (requires λm₀ → live m₀ output ∧ len = length output)
  (ensures λm₀ success m₁ → modifies output m₀ m₁ ∧
   (success ⇒ valid_from parse_example m₁ output 0))
  if len < 8 then false (* output buffer too small *)
  else let off₁ = write_u32 output 0 left in let _ = write_u32 output off₁ right in
    let mem = get () in example_intro mem output 0; true

In this example, the user needs to reason about the concrete byte offsets: they need to provide the positions where values should be written, relying on write_u32 returning the position just past the 32-bit integer it wrote in memory. Then, they have to apply the validity lemma: satisfying its precondition involves (crucially) proving that the writing of the second field write does not overlap the first one, through Low* memory framing. These proofs are here implicit but still incur verification cost to the SMT solver; as the complexity of the structs increases, it has a significant impact on the SMT proof automation. Moreover, the user also needs to worry about the size of the output buffer being large enough to store the two integer fields.

4.1 The Write Layered Effect

To abstract low-level byte layout and error handling, we define a Write effect layered over Stack, where an effectful computation f: Write t pbefore pafter returns a value of type t while working on a hidden underlying mutable buffer. Each such computation requires upon being called that the buffer contains binary data valid according to the pbefore parser specification, and ensures that, if successful, it contains binary data valid according to pafter on completion. Thus, such a Write computation presents the user with a simple parser-indexed parameterized state and error monad, in such a way that memory safety, binary layout and error propagation details are all hidden away from the user at once. Returning to the example above, we can define:

(Elementarily write an integer *)

val write_u32: U32.t → Write unit parse_empty parse_u32

(A generic higher–order framing operator, to be able to write two pieces of data in a row *)

val frame (#t: Type) (#pframe #pafter : parser)(f: unit → Write t parse_empty pafter)
 : Write t pframe (parse_pair pframe pafter)
(∗ A lifting of the serialization lemma, with all details on binary layout hidden. Computationally a no-op. ∗)

val write_example_correct : unit → Write unit (parse_pair parse_u32 parse_u32) parse_example

This last lemma states that, if the output buffer contains valid data for parsing a pair of two integers, then calling this function will turn that data into valid data for parsing an example struct value. With those components, the user can now write their formatting code more succinctly, as shown below. The output buffer, offsets and error propagation are hidden in the effect and so the user no longer needs to explicitly reason about them. Furthermore, the code becomes much more self-explanatory.

let write_example (x1 x2 : U32.t) : Write unit parse_empty parse_example = write_u32 x1; frame (λ _ → write_u32 x2); write_example_correct()

Optimizing Verification. Thanks to the abstraction offered by the Write effect, it is possible to spare the user from even more mundane details, such as having to explicitly call write_example_correct.

We can embed parser rewriting rules in the subsumption rule for Write. In other words, if \(\text{repr t pbefore pafter}\) is the type of the underlying representation of a Write computation, then we can define a subcomp rule for automatic rewriting of the pafter parser:

\[
\text{val subcomp (t: \textit{Type}) (p1 p2 p\textsuperscript{'}: \textit{parser}) (r: \textit{repr t p1 p2}): \textit{Pure} (\textit{repr t p1 p\textsuperscript{'}})
\]

\[
\text{(requires (valid_rewrite p2 p\textsuperscript{'})) (ensures (λ _ → T))}
\]

where valid_rewrite p2p\textsuperscript{'} is a relation on parser specifications stating that any binary data valid for p2 is also valid for p\textsuperscript{'}.

Since the valid_rewrite goals can automatically be solved via SMT, this allows us to rewrite our example as simply \(\text{write_u32 x1; frame (λ _ → write_u32 x2)}\). The only overhead that remains is framing, which we aim to automate using the techniques of §5.2.

Representing Write: A peek beneath the covers. The representation of Write is interesting, involving a dependent pair of indexed monads, where p.datatype is the type of the values parsed by p.

\[
\text{type write (t: \textit{Type}) pbefore pafter = (spec: write_spec t pbefore.datatype pafter.datatype & write_impl t pbefore pafter spec)}
\]

The first field, spec, is a specational parameterized monad evolving an abstract state from pbefore.datatype to pafter.datatype. The second field is the Low∗ implementation, indexed by a pair of parsers and the spec. As such, write_impl is a parameterized-mond-indexed monad, or a form of parameterized Dijkstra monad, a novel construction, as far as we are aware.

To compile a Write computation to C, or call it from other Low∗ code, we simply reify it and project its Low∗ implementation:

\[
\text{let reify_spec \# t \# pbefore \# pafter (f: unit → Write t pbefore pafter) : write_spec t pbefore.datatype pafter.datatype = (reify (f ()).1)}
\]

\[
\text{(∗ Extract the Low∗ code of a computation to compile it to C ∗)}
\]

\[
\text{let reify_impl \# t \# pbefore \# pafter (f: unit → Write t pbefore pafter) : write_impl t pbefore pafter (reify_spec f) = (reify (f ())).2}
\]

4.2 Application: TLS 1.3 Handshake Extensions

We have applied the Write effect to generate the list of extensions of a TLS 1.3 [Rescorla 2018] ClientHello handshake message, that a client sends to a server to specify which cipher suites and other protocol extensions it supports. This is the most complex part of the handshake message format, involving much more than just pairs: it involves variable-sized data and lists prefixed by their size in bytes, as well as tagged unions where the parser of the payload depends on the value of the tag. Our implementation of ClientHello messages written using the Write effect compiles to
C and executes; we are currently integrating it into a low-level rewriting of an implementation of TLS in F* [Bhargavan et al. 2013].

As an excerpt of this example, we show how to write a variable-sized list of 32-bit integers, the list being prefixed by its size in bytes. The TLS 1.3 RFC [Rescorla 2018], specifying the data format description for handshake messages, bounds the size of every such list, excluding the size header, independently of the size of the actual output buffer. If \( p \) is a parser for the elements of the list, then \( \text{parse\_vllist } p \ \text{min} \ \text{max} \) is a parser that first reads an integer value that will be the storage size of the list in bytes, checks that it is between \( \text{min} \) and \( \text{max} \), then parses the list of elements using \( p \) for each. Thus, the following code excerpt writes a list of two integers into the output buffer:

\[
\text{let write\_two\_ints } (\text{max\_list\_size}: \text{U32}\ t) := \\
\text{write\_vllist\_nil } (\text{parse\_u32 } 0 \ \text{max\_list\_size}); \\
\text{frame } (\lambda \_ \rightarrow \text{write\_u32 } 18\ul); \\
\text{extend\_vllist\_snoc } (); \\
\text{frame } (\lambda \_ \rightarrow \text{write\_u32 } 42\ul); \\
\text{extend\_vllist\_snoc } ()
\]

Following the TLS 1.3 RFC, the value of \( \text{max\_list\_size} \) constrains the size of the size header, but thanks to the abstraction provided by \( \text{Write} \), the user does not need to know about that actual size. Indeed, that code relies on two combinators:

\[
\text{val write\_vllist\_nil } (p\ : \text{parser}) (\text{max}\ : \text{U32}\ t) := \\
\text{Write unit } \text{parse\_empty } (\text{parse\_vllist } p \ 0 \ \text{max}); \\
\text{val extend\_vllist\_snoc } (#p\ : \text{parser}) (#\text{min} \ #\text{max}: \text{U32}\ t) ()
\]

\( \text{write\_vllist\_nil} \) actually starts writing an empty list by writing 0 as its size header. Then, \( \text{extend\_vllist\_snoc} \) assumes that the output buffer contains some variable-sized list immediately followed by an additional element and “appends” the element into the list by just updating the size header of the list; thus, the new element is not copied into the list, since it is already there at the right place. \( \text{extend\_vllist\_snoc} \) also dynamically checks whether the size of the resulting list is still within the bounds expected by the parser, and exits with error if not.

A more powerful version of the \( \text{write} \) effect with support for Hoare-style pre- and postconditions to prove functional correctness properties on the actual values written to the output buffer, as well as error postconditions, in addition to correctness wrt. the data format, is underway. With such an enhanced version, we leverage pre- and postconditions to avoid dynamic checks on writing variable-size list items.

5 CASE STUDY: STRUCTURING A DSL FOR SEPARATION LOGIC

Steel [Swamy et al. 2020] is a shallow embedding of concurrent separation logic (CSL) in F*, with support for atomic computations and dynamically allocated invariants. In this section, we show how we use layered effects to provide a DSL for Steel better suited for practical verification. We first show how to seamlessly interoperate between non-atomic, atomic, and ghost computations using indexed effects. We then demonstrate how we use polymonads to automate frame inference in Steel, and re-implement a library for fork/join concurrency. Saving the programmer from CPS’ing their code, we build a final layered effect for continuations on top of Steel.
of our construction to provide a Unix-like interface
for fork and join, which also benefits from a high-
level of automation for framing through polymonads. We summarize the scaffolding of effects we
use in the figure alongside. Each effect appears in two forms, an $\alpha$-form for “annotated”, and an
$F$-form for “framed”—the paired structure is in support of frame inference. Simple lines represent
binds, dashed lines represent subsumption relations, and dotted lines correspond to sub-effects.

5.1 Effects to Structure a Sub-Language of Atomic and Ghost Computations

Steel enables specification of pre- and postconditions of functions using extensible separation logic
assertions of type $\text{slprop}$. Steel functions have their own effect, a Hoare monad parameterized by a
return type $a$, a precondition $fp$, and a postcondition $fp'$ that depends on the return value.\footnote{In practice, the Steel effect contains two additional indices for heap predicates, enabling specifications in the style of implicit dynamic frames. We omit these indices in this paper: these predicates are always trivial in the examples we present.} The
program logic is designed for partial correctness, and relies on $F^{*}$’s existing support for divergence
to model general recursive, potentially deadlocking Steel programs—the divergent annotation on the
$\text{effect}$ definition below indicates this.

\begin{verbatim}
let steel (a:Type) (fp:slprop) (fp':a -> slprop) = (* elided, inherited from Swamy et al. [2020] *)
let bind (f:steel a fp0 fp1):steel b fp0 fp2 = ...

divergent effect { Steel (a:Type) (fp:slprop) (fp':a -> slprop) = steel with bind; ... }
\end{verbatim}

To reason about concurrent programs, the Steel program logic provides a model of total atomic
actions, as well as dynamically allocated invariants that can be freely shared between threads,
similarly to other CSL-based frameworks like Iris [Jung et al. 2018]. Invariants can only be safely
accessed and restored by atomic actions. To assist with verification, Steel also supports a notion of
ghost state, as well as ghost, computationally irrelevant computations. The composition of a ghost
action with an atomic action is also deemed atomic, since ghost code does not actually execute; but
two non-ghost actions cannot be composed atomically.

To distinguish atomic and non-atomic code, we define a new effect, $\text{SteelAtomic}$.\footnote{If $\text{atomic}$ is partial: the safe composition of ghost
bind $\text{atomic}$ is partial: the safe composition of ghost
and atomic computations is captured in the refinement on $g_2$; to compose two atomic computations,
one of them has to be ghost.}

\begin{verbatim}
let atomic (a:Type) (u:inames) (g:bool) (fp:slprop) (fp':a -> slprop) = ... 
let bind_atomic #a #b #u #fp0 #fp1 #fp2 #g1 (#g2:bool) (g1 || g2)
  (f:atomic a u fp0 fp1) (g:x:a) -> atomic b u g2 (fp1 x) fp2) = atomic b u (g1 && g2) fp0 fp2 = ...

effect { SteelAtomic (a:Type) (u:inames) (g:bool) (fp:slprop) (fp':a -> slprop) = atomic with ... }
\end{verbatim}

SteelAtomic enhances the $\text{Steel}$ effect with two additional indices: a set of currently opened
invariants $u$, and a boolean $g$ indicating whether the computation is ghost. The type $\text{inames}$ is an
abbreviation for a set of invariant names. Note, $\text{bind}\_atomic$ is partial: the safe composition of ghost
and atomic computations is captured in the refinement on $g_2$; to compose two atomic computations,
one of them has to be ghost.

Using this effect, we can then expose the atomic opening and closing of an invariant as provided
by the program logic. Given an invariant $i$ protecting a separation logic assertion $p$, an atomic
computation $f$ is allowed to access $p$ as long as $p$ is restored once the computation terminates.
Additionally, $i$ must not be in the set of currently opened invariants $u$ to prevent recursive opening
of invariants. In a sense, $\text{with}\_inv$ parallels to effect handling in \S 2.2. Given $f$ which uses invariants
$i \cup u$, $\text{with}\_inv$ handles $i$, leaving a computation that only uses invariants $u$.

\begin{verbatim}

val with_inv (i:inv p) (f:unit -> SteelAtomic a (i \cup u) g (p * q) (l x -> p * r x)) = SteelAtomic a u g q r
\end{verbatim}

Since atomic computations are a subset of computations, $\text{SteelAtomic}$ is a sub-effect of the
generic $\text{Steel}$ effect. Any $\text{SteelAtomic}$ computation with no opened invariant can be lifted to a
Steel computation. The use of sub-effecting enables a smooth interoperation between the two

let atomic (a:Type) (u:inames) (g:bool) (fp:slprop) (fp':a -> slprop) = ... 
let bind_atomic #a #b #u #fp0 #fp1 #fp2 #g1 (#g2:bool) (g1 || g2)
  (f:atomic a u fp0 fp1) (g:x:a) -> atomic b u g2 (fp1 x) fp2) = atomic b u (g1 && g2) fp0 fp2 = ...

effect { SteelAtomic (a:Type) (u:inames) (g:bool) (fp:slprop) (fp':a -> slprop) = atomic with ... }
\end{verbatim}

SteelAtomic enhances the $\text{Steel}$ effect with two additional indices: a set of currently opened
invariants $u$, and a boolean $g$ indicating whether the computation is ghost. The type $\text{inames}$ is an
abbreviation for a set of invariant names. Note, $\text{bind}\_atomic$ is partial: the safe composition of ghost
and atomic computations is captured in the refinement on $g_2$; to compose two atomic computations,
one of them has to be ghost.

Using this effect, we can then expose the atomic opening and closing of an invariant as provided
by the program logic. Given an invariant $i$ protecting a separation logic assertion $p$, an atomic
computation $f$ is allowed to access $p$ as long as $p$ is restored once the computation terminates.
Additionally, $i$ must not be in the set of currently opened invariants $u$ to prevent recursive opening
of invariants. In a sense, $\text{with}\_inv$ parallels to effect handling in \S 2.2. Given $f$ which uses invariants
$i \cup u$, $\text{with}\_inv$ handles $i$, leaving a computation that only uses invariants $u$.

\begin{verbatim}

val with_inv (i:inv p) (f:unit -> SteelAtomic a (i \cup u) g (p * q) (l x -> p * r x)) = SteelAtomic a u g q r
\end{verbatim}

Since atomic computations are a subset of computations, $\text{SteelAtomic}$ is a sub-effect of the
generic $\text{Steel}$ effect. Any $\text{SteelAtomic}$ computation with no opened invariant can be lifted to a
Steel computation. The use of sub-effecting enables a smooth interoperation between the two
5.2 Controlling Frame Inference with a Polymonad

The bind function presented in section 5.1 requires the postcondition of the first computation to match exactly with the precondition of the second computation. If the two computations operate on different resources, it forces a programmer to reason manually about framing each resource. Such reasoning becomes quickly intractable, as it is required for each composition. Thus, automating framing is unavoidable in a separation logic-based verification framework.

To this end, one possible approach is to incorporate framing into every bind. Let us consider two functions \( f : \text{unit} \rightarrow \text{SteelF} p (\lambda_\_ \rightarrow q) \) and \( g : \text{unit} \rightarrow \text{SteelF} r (\lambda_\_ \rightarrow s) \). Then, for any \( \text{slprops} \, fp \) and \( \text{fp}' \), we can apply framing to typecheck \( f () ; g () \) as \( \text{SteelF} (p \cdot fp)(\lambda_\_ \rightarrow s \cdot fp') \) as long as the framed postcondition of the first computation, \( q \cdot \text{fp} \), is equivalent to the framed precondition of the second computation, \( r \cdot \text{fp}' \). Applying framing at every bind, the number of frames in the specification will grow linearly with the number of compositions. If the frames are given, one can partially decide equivalence by associativity-commutativity rewriting in the commutative monoid (\( \text{slprop} \cdot \), \( \text{emp} \)). But automatically inferring them becomes tricky: when given an equivalence with more than one frame to infer, several non-equivalent solutions might be admissible. If for instance \( p \cdot fp \cdot fp' \) is equivalent to \( p \cdot q \) with \( fp \) and \( fp' \) to be inferred, then both \( fp = q \), \( fp' = \text{emp} \) and \( fp = \text{emp} \), \( fp' = q \) are valid solutions.

The approach we propose instead is to apply the frame rule as little as possible. If a computation has already been framed, we do not frame it again when composing it. Then, each pre- and postcondition contains at most one frame to be inferred. To distinguish framed and not-yet-framed computations, we present a polymonad based on two effects \( \text{SteelF}, \text{SteelA} \), representing computations types that are always in the form \( \text{SteelF} \, a (\text{pre} \leftrightarrow \text{?f}) \, \lambda x \rightarrow \text{post} x \leftrightarrow \text{?f} \), where \( \text{?f} : \text{slprop} \) is a frame metavariable to be solved; and \( \text{SteelA} \), a computation type with no metavariables. Although both effects use the same underlying representation, composing a \( \text{SteelF} \) computation \( f \) with a \( \text{SteelA} \) computation \( g \) yields a \( \text{SteelF} \) computation, effectively adding a frame to \( g \). To control the addition of frame variables around \( \text{SteelA} \) computations, we use polymonadic binds (§3.2), as shown below.

```
val bind_steelf_steela (split_post_f:squash (\( x \cdot \text{post} x \leftrightarrow \text{pre}_g x \leftrightarrow \text{frame} \))
        (f: \text{steel a pre}_f \text{post}_f) (g: \lambda x \rightarrow \text{steel b pre}_g \text{post}_g) : \text{steel b pre}_f (\lambda y \rightarrow \text{post}_g y \leftrightarrow \text{frame})

polymonadic_bind (\text{SteelF, SteelA}) \rightarrow \text{SteelF} = \text{bind_steelf_steela}
```

The main point of interest here is the introduction of the additional variable \( \text{frame} \) and the \text{split_post_f} hypothesis, which requires proving that the postcondition of \( f \) (i.e., \text{post}_f x \) can be split into the precondition of \( g \) (i.e., \text{pre}_g x \) and a \text{frame}, which is propagated across \( g \) and appears in the postcondition, along with \text{post}_g y.\(^6\) Compare this to the \text{bind} presented in §5.1, which requires \text{post}_f and \text{pre}_g to match exactly, without introducing a frame. The polymonadic bind (\text{SteelA, SteelF}) \rightarrow \text{SteelF} adds a frame around the first computation; (\text{SteelA, SteelA}) \rightarrow \text{SteelF} frames both; (\text{SteelF, SteelF}) \rightarrow \text{SteelF} frames neither. Using this polymonad, we ensure that each \text{slprop} appearing in a pre- or postcondition has at most one frame to be inferred. Exploiting this invariant, we design a (partial) decision procedure for frame inference as an \( F^* \) tactic, to which we defer the resolution of all frame variables and goals like \text{split_post_f}.

From a programmer’s perspective, all Steel functions appear to have a \text{SteelA} computation type, which ensures that they can be called and framed from any context in which the precondition is satisfied. \text{SteelF} is but an internal artifact that provides a clean design to automate frame inference.

\(^6\)We could use a separating implication \( \rightarrow \) instead of \( \sim \), though that would allow resources to be dropped silently.
As any composition leads to a SteelF computation, the last missing piece is a way to automatically retype a SteelF computation as SteelA in order to match a user-provided annotation—this is achieved below, by lifting and subsuming SteelF to SteelA in a single step. The polymonadic constructs provide the control needed to orchestrate frame inference.

val subcomp_steelf_steela(_::squash (pre_g ~ pre_f)) (_::squash (\x. post_f x ~ post_g x))
(f::steel a pre_f post_f) : steel a pre_g post_g

polymonadic_subcomp SteelF <- SteelA = subcomp_steelf_steela

This strategy of adding frames isn’t specific to the Steel effect: we augment the SteelAtomic effect with an identical construction—we believe it would be also be well-suited for other program logics that involve framing, such as the formatters from §4. We leave this as future work.

Using this approach, we re-implement several existing Steel libraries. We show below our implementation of fork. It relies on par, the basic concurrency primitive provided by Steel for structured parallelism. Using automated framing, the programmer no longer needs to call frame, or the ghost h_intro_emp_r and h_elim_emp_l, yielding just let t = new_thread q in par (spawn f t) (g t) and eliminating all proof clutter. The crossed out lines from the original code below are no longer necessary.

val par (f:unit → SteelA aL preL postL)  (g:unit → SteelA aR preR postR)
: SteelA (aL & aR) (preL * preR) (\l xL xR → postL xL * postR xR)

let fork (f: (unit → SteelA unit p (\_ → q))) (g: (thread q → unit → SteelA unit r (\_ → s)))
: SteelA unit (p * r)(\_ → s) = h_intro_emp_l (p \star r+)
let t = frame (fun \_ → new_thread q) (p \star r+) in
h_elim_emp_l (p \star r+)
par (spawn f t) (g t)
h_elim_emp_l

5.3 Layering Continuations

The fork function presented above expects both the parent and child threads as arguments and blocks until they both return. In this section, we show how layering an additional effect, SteelKA, on top of SteelA allows us to provide a more idiomatic fork/join by capturing the continuation of the parent thread within an effect. We begin with a steelK representation type of CPS functions into SteelA, indexed by pre and post conditions. Each steelK function is frameable and parametric in the final postcondition, but the final answer type is fixed to unit.

let steelK (a:Type) (pre: slprop) (post:a → slprop) =
#frame:slprop → #postf:slprop → f:(x:a → SteelA unit (frame * post x) (\_ → postf)) →
SteelA unit (frame * pre) (\_ → postf)

effect { SteelKA (a:Type) (pre: slprop) (post: a → slprop) = steelK with ... }

Given SteelKA, we can provide a more direct signature for fork, taking only the function to be spawned and returning a handle for the child thread. Its implementation basically calls the previous fork passing the current continuation as the parent thread. One can then use join to wait for the child thread, and obtain its postcondition q.

val fork (## #q : slprop) (f: unit → SteelKA unit p (\_ → q)) : SteelKA (thread q) p (\_ → emp)
val join (#q : slprop) (t : thread q) : SteelKA unit emp (\_ → q)

By replicating our framing methodology for SteelKA (by introducing a sibling effect of framed computations SteelKF), and lifting SteelA to SteelKA, we now have a C-like fork/join:
let example (r : ref int) : SteelKA (ref int) (pts_to r 0) (\x \rightarrow pts_to r 1 * pts_to x 2) =
let pl = kfork (\_ \rightarrow write r 1) in let x = alloc 2 in kjoin pl; x

In this small example, we initially take a reference to an integer as an argument, which we assume to contain the (unused) value 0. We then fork, performing a write in the child thread while we allocate a new reference in the parent thread. Both write and alloc are SteelA functions that are automatically lifted to SteelKA.

6 RELATED WORK & CONCLUSIONS

We have discussed several strands of related work throughout the paper. We focus here on relating our work to two main themes not discussed in detail elsewhere.

Unifying frameworks for effectful programming and semantics. Given the variety of monad-like frameworks for effects, a unifying theory that accounts for all the variants is a subject of some interest. Tate’s (2013) producers and, equivalently, Hicks et al.’s (2014) polymonads are attempts at subsuming frameworks, Tate focusing more on the semantics while Hicks et al. consider programmability, with Bracker and Nilsson [2015] even providing a Haskell plugin for polymonad programming. However, both frameworks cater to simply typed programs, and as such do not account for inherently dependently typed constructs such as Hoare and Dijkstra monads. Orchard et al. [2020] propose to unify graded and parameterized monads by moving to category-indexed monads, studying them from a semantic perspective only, while also working in a simply typed setting. Orchard and Petricek [2014] also propose a library to encode effect systems with graded monads in Haskell. As far as we are aware, we are the first to provide a programming framework that encompasses monad-like structures with arbitrary indexing in a dependently typed setting, demonstrating its value for programming and proving.

Programming and proving with algebraic effects. Our library for algebraic effects is perhaps related most closely to Brady’s (2013) Effects DSL in the dependently typed language Idris. The main points of difference likely stem from what is considered idiomatic in Idris versus F★. For instance, the core construct in the Idris DSL is a type indexed by a list of effects (similar to our tree a l)—whereas in Idris the indexing is intrinsic, our trees are indexed extrinsically with a refinement type, enabling a natural notion of subsumption on indices based on SMT-automated effect inclusion. Idris’ core effects language is actually a parameterized monad—our supplement and full version of the paper show a similarly parameterized version of our tree type. More specifically, by packaging our trees into an effect, we benefit from F★’s monadic elaboration, avoiding the need for monadic syntax, idiom brackets and the like, with implicit subsumption handled by SMT. Further, unlike Brady, we provide a way to interpret Read-Write trees into a Dijkstra monad, enabling functional correctness proofs. While have only taken initial steps in this direction, we appear to be the first to actually verify stateful programs in this style. Maillard et al. [2019] propose a tentative semantics to interpret algebraic effect handlers with Dijkstra monads, and use their approach to extrinsically verify the totality of a Fibonacci program with general recursion. Our work builds on theirs, requires fixing the interpretation of the operations, but yields a methodology to do a proof of stateful programs. Besides, with layered indexed effects, we get to choose whether to work with Dijkstra monads or not—in contrast, Maillard et al.’s framework cannot support the list-of-effects indexed graded monad. Algebraic effects have also been embedded in Haskell in several styles, notably by Kiselyov and Ishii’s (2015) “freer” monads, relying on encodings of dependent types in Haskell’s type system to also index by a list of effect labels, while focusing also on efficient execution, a topic we have not yet addressed for our Alg effect.
Conclusions. Embracing the diversity of indexed monad-like constructions, and aiming to reap their benefits when programming with effects in a dependently typed language, we have designed and implemented layered indexed effects as a feature of F*. The gains are manifold, simplifying F*'s core logic, enabling new abstractions for programming and proving, and simplifying the construction of effectful programs. Despite several significant case studies using our new system, we feel we have just scratched the surface—the possibilities of mixing a variety of new effect disciplines to better structure programs and their proofs seem endless.

REFERENCES


