



HAL
open science

Programming and Proving with Indexed Effects

Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro,
Nikhil Swamy

► **To cite this version:**

Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, Nikhil Swamy. Programming and Proving with Indexed Effects. Microsoft Research. 2020. hal-04256360

HAL Id: hal-04256360

<https://inria.hal.science/hal-04256360v1>

Submitted on 24 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Programming and Proving with Indexed Effects

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

Proving properties about effectful programs is hard. New application-specific abstractions based on indexed monads can help simplify programming and proving. However, existing languages lack support to develop and use such abstractions.

The main contribution of this paper is a type-and-effect system that enables program proof developers to design new effect-typing disciplines based on indexed monads, making proofs simpler and more abstract and allowing programs to be developed in a direct, applicative syntax while automatically elaborating them into a core language of pure, total functions where the monadic structure is made explicit.

We have implemented our system as a new feature in the F^* programming language, enhancing its existing user-defined effect system to cover *all* forms of indexed monads. In doing so, we have also simplified the core language of F^* , allowing us to derive basic Dijkstra monad constructions in F^* that were previously primitive.

Finally, we present several case studies developing new indexed monad constructions to structure program proofs in settings including information flow control, algebraic effects, and low-level binary format parsers.

1 INTRODUCTION

Xavier Leroy, in his Royal Society Milner Award lecture,¹ claims that purely functional programming is the shortest path to writing and proving a program. Few practitioners of program proofs will disagree—programming and reasoning in the presence of computational effects is hard. However, effectful programming is indispensable in many domains, e.g., when building low-level or high-performance software. We are interested in techniques that simplify the construction of proofs of correctness and security of effectful programs, starting by representing effects using *monads*.

While Moggi [1989] and Wadler [1992] firmly established monads as both a categorical and programmatic basis on which to develop effectful programs, proving the correctness of programs with monadic effects is somewhat less settled. Several researchers have proposed variants of monads, with richer indexing structure, in support of more precise static reasoning. Many of these proposals have been profitably used in a variety of settings, yet no single proposal has emerged as universal—given the diversity of program reasoning tasks, universality of structures in support of reasoning is hard to expect or imagine. Rather, we embrace the diversity of indexed monads and seek to use them to structure and simplify program proofs. We briefly survey the landscape.

Monads. From a programmer’s perspective, a monad M is a typeclass representing an effectful computation, supporting the following two combinators: `return`: $a \rightarrow M a$, to promote a pure value to an M computation; and `bind`: $M a \rightarrow (a \rightarrow M b) \rightarrow M b$, to sequentially compose two M -computations, where `bind` is associative and `return` is both a left and right unit of `bind`. A great many computational effects have been shown to be expressible as monads, including state, exceptions, continuations, parsing, printing, asynchrony, and many others besides. However, the type $M a$ is relatively uninformative. A program with type $M a$ may exhibit any of the effects encoded in M when executed, e.g., it may read and write the state arbitrarily. When trying to prove a program correct, this imprecision is usually unacceptable, leading to several indexed monad structures with adaptations of the monad laws to account for the indices.

¹<https://xavierleroy.org/talks/Milner-award-lecture.pdf>

50 *Graded monads.* Katsumata’s (2014) graded monads are a monad-like typeclass G indexed by
 51 a monoid $\{I, \oplus, \epsilon\}$, whose return and bind have the following signatures: $\text{return} : a \rightarrow G a \epsilon$ and
 52 $\text{bind} : G a i \rightarrow (a \rightarrow G b j) \rightarrow G b (i \oplus j)$. By choosing the indexing monoid carefully, one can recover
 53 some precision in static reasoning. For instance, the indices can be used to constrain the mem-
 54 ory locations a computation may write, ensuring, depending on the index i , that a given $G a i$
 55 computation leaves certain parts of memory unchanged.

56 *Parameterized monads.* Atkey’s (2009) parameterized monads are a typeclass A with two indices,
 57 with the following combinators: $\text{return} : a \rightarrow A a p p$ and $\text{bind} : A a p q \rightarrow (a \rightarrow A b q r) \rightarrow A b p r$. Given
 58 an $e : A a p q$, the index p is an abstraction of the resources expected by e , and q abstracts the resources
 59 remaining after e executes. Parameterized monads have been used to reason about a variety of
 60 effects, including, for example, message passing programs using session types.

61
 62 *Hoare monads.* Working in a dependently typed setting, and aiming initially to prove stateful
 63 programs correct in Coq, Nanevski et al. [2008] developed the Hoare monad, a typeclass H , in-
 64 dexed by memory predicates $p : \text{mem} \rightarrow \text{prop}$ and $q : a \rightarrow \text{mem} \rightarrow \text{prop}$, with combinators of the form
 65 $\text{return} : x : a \rightarrow H a (p x) p$, meaning that a pure computation returns an $x : a$ while preserving any pred-
 66 icate $p x$ on the state; and $\text{bind} : H a p q \rightarrow (x : a \rightarrow H b (q x) r) \rightarrow H b p r$, a combinator whose indexing
 67 structure represents the rule for sequential composition in Hoare logics. Encoding a Hoare logic in
 68 the indices is a powerful concept, and Hoare monads have been used to prove the correctness of
 69 many programs in a variety of program logics. But, even among Hoare monads, several variants
 70 exist. For example, in some versions, the postcondition q is a relation on a pair of memory states,
 71 i.e., $q : \text{mem} \rightarrow a \rightarrow \text{mem} \rightarrow \text{prop}$.

72
 73 *Dijkstra monads.* Seeking to compute verification conditions for programs with effects beyond
 74 just state, Swamy et al. [2013] proposed Dijkstra monads. Refined further by Swamy et al. [2016],
 75 Ahman et al. [2017] and Maillard et al. [2019], a Dijkstra monad D is a monad-like typeclass
 76 where the indexing structure is itself a monad $\{M, \text{return}, \text{bind}\}$. That is, D has the following combi-
 77 nators: $\text{return} : x : a \rightarrow D a (M.\text{return } x)$, and $\text{bind} : D a m \rightarrow (x : a \rightarrow D b (n x)) \rightarrow D b (M.\text{bind } m n)$. Intuitively,
 78 the computational monad D is abstracted by the specification monad M , with a morphism between
 79 the two encoded in the indexing structure. Dijkstra monads are at the core of the F^* programming
 80 language [Swamy et al. 2016] and have been used in the verification of several large develop-
 81 ments [Bhargavan et al. 2017].

82 83 84 1.1 New hybrid constructions

85 A central observation of this paper is that while each of these indexed monad structures offer
 86 reasoning principles for effectful programs on their own, using them in combination yields a
 87 multitude of other structures that can help in producing simpler, more structured proofs of effectful
 88 programs. We present new hybrid constructions involving graded Hoare monads, parameterized
 89 Dijkstra monads, graded Dijkstra monads, graded doubly-Hoare monads, and other such hybrid
 90 structures, exploiting them to simplify the proofs of programs ranging from the correctness of
 91 binary format serializers to information flow control.

92 A starting point to effectively exploit these exotic indexed structures for programming and
 93 proving is a unified, programmable framework supporting them all. The syntactic overhead of
 94 programming directly with monadic structures is prohibitive—consider that programming with
 95 even regular monads is tedious absent Wadler’s classic *do*-notation. However, whereas all monadic
 96 programs benefit from the *do*-notation, other indexed monads, not being instances of the monad
 97 typeclass, do not enjoy such benefits. E.g., in Haskell, parameterized monads are captured by
 98

the Monadish typeclass,² for which no special syntax is available. Given the diversity of indexing structures we wish to use, a single typeclass to cover them all is infeasible.

1.2 Type-and-effect directed elaboration

Lacking a typeclass for our structures, we develop a new language feature to support a type-and-effect directed elaboration of source programs written in a direct, applicative syntax into any indexed monad structure. Our feature, *indexed effects*, is usable with any monad-like type constructor L , with an arbitrary number of indices, and combinators $\text{return} : x:a \rightarrow L a \vec{i}$ and $\text{bind} : L a \vec{i} \rightarrow (x:a \rightarrow L b \vec{j}) \rightarrow L b \vec{k}$ —note the indices may vary arbitrarily in return and bind . Given such a signature, our algorithm allows programs to be developed in an ML-like applicative syntax, while elaborating them automatically into the underlying monad-like combinators on L . We have implemented indexed effects in F^* , enhancing its user-defined effect system, previously limited to Dijkstra monads only, to cover all forms of indexed monads.

For a first example, consider the graded monad $\text{gst} (a:\text{Type}) (t:\text{tag})$, with $\text{tag} = R \mid RW$ shown below, with a refinement type to state that read-only computations do not modify the state, and with actions to read and write the state.

```
let state = int    type tag = R | RW    let ( $\oplus$ ) t0 t1 = match (t0, t1) with | (R, R)  $\rightarrow$  R | _  $\rightarrow$  RW
let gst (a:Type) (t:tag) = s $\emptyset$ :state  $\rightarrow$  r:(a & state) { t=R  $\implies$  s $\emptyset$  == snd r }
let return (x:a) : gst a R =  $\lambda$ s  $\rightarrow$  x,s
let bind (f:gst a t0) (g: a  $\rightarrow$  gst b t1) : gst b (t0  $\oplus$  t1) =  $\lambda$ s $\emptyset$   $\rightarrow$  let x, s1 = f s $\emptyset$  in g x s1
let read () : gst state R =  $\lambda$ s  $\rightarrow$  s,s    let write (s:state) : gst unit RW =  $\lambda$ _  $\rightarrow$  (), s
```

To increment the state, one would write $\text{bind} (\text{read}()) (\lambda x \rightarrow \text{write} (x + 1))$ and many dependent type systems could infer the type gst unit RW . For such a simple program, this may seem adequate. However, as the indices become richer, explicitly monadic programming can be an obstacle.

With our new support for user-defined indexed effects in F^* , we can turn the gst monad into a new indexed *computation type* GST, while also indicating to the system to implicitly re-index types when needed, e.g., in the branches of conditional computations.

```
let subcomp (f:gst a t0 {  $\exists$ t. t1 == t0  $\oplus$  t }) : gst a t1 = f
let if_then_else (f:gst a t0) (g:gst a t1) (b:bool) = gst a (t0  $\oplus$  t1)
effect { GST (a:Type) (t:tag) with { repr = gst; return; bind; read; write; subcomp; if_then_else } }
```

With these definitions in place, one can write $\text{if } b \text{ then } (\text{write} (\text{read} () + 1); 0) \text{ else } \text{read}()$, while the framework infers the computation type GST int RW and internally elaborates the program into the following explicitly monadic form:

```
if b then subcomp (bind (read()) ( $\lambda$  x  $\rightarrow$  bind (write (x + 1)) ( $\lambda$  _  $\rightarrow$  return 0))) else subcomp (read())
```

Effect definitions can also be layered, e.g., we could add a layer to represent exceptions on top of the GST effect, with implicit coercions to move between them.

1.3 Formalization of indexed effects and simplification to the theory of F^*

To formalize our system, we design Indexed Monadic F^* (IMF *), a surface language with user-defined indexed effects, and a simple type-and-effect directed elaboration of IMF * programs into TotalF * , a core lambda calculus with dependent and refinement types. Our main theorem proves that the translation from IMF * to TotalF * is well-typed (§3).

Prior to our work, the core calculus of F^* included a primitive notion of Dijkstra monads [Ahman et al. 2017; Swamy et al. 2016]. Indeed, all other Dijkstra monads in F^* built upon this primitive notion. With IMF * , Dijkstra monads can be defined as just another indexed effect and need no

²<https://hackage.haskell.org/package/twilight-stm-1.2/docs/Control-Concurrent-STM-Monadish.html>

longer be primitive. As result, not only does our work add support for programming with rich indexing structures in F^* , but it also simplifies the core logical underpinnings of F^* to just $\text{Total}F^*$. Simplifying the core is a significant advancement for a proof assistant.

1.4 Applications of the new hybrid constructions

We present three case studies of indexed effects at work. The first is a new graded, Hoare monad for information flow control and functional correctness of stateful programs (§2). Next, we show a library of generic algebraic effects and handlers, using a graded Dijkstra monad to verify heap-manipulating programs in this framework (§4). Finally, we improve support for low-level message formatting in EverParse [Ramananandro et al. 2019], a library of monadic parser and formatter combinators. By layering two parameterized monad-indexed monads on top of EverParse’s existing use of a Hoare monad for C programs, we obtain more concise programs and better proof automation, while yielding verified C code devoid of performance overhead (e.g., due to intermediate allocations and copies) inherent in the purely functional formatters developed previously (§5)³.

Separately, providing evidence of the usefulness of our work at scale, indexed effects in F^* have already been used extensively in Steel [Fromherz et al. 2021; Swamy et al. 2020], a dependently typed, concurrent separation logic shallowly embedded in F^* , using an indexed monad with six indices to capture various components of Steel’s logic. Additionally, Bhargavan et al. [2021] use indexed effects in F^* to reason about properties of a global, interleaved execution trace of cryptographic protocol sessions, using it at the core of a system that partially automates symbolic proofs of cryptographic security (§6).

In all these cases, effect indices provide an abstraction to reason about effectful programs. When these indices come from familiar algebraic structures like monoids and monads, proofs of effectful programs can be reduced to purely functional programming, following Xavier Leroy’s guidance.

The diversity of our experience encourages us to conclude that richly indexed effects, coupled with simple language support for elaboration, allows program proof developers to craft new abstractions and benefit from simpler proofs, while also enjoying a direct programming style with automatic inference and elaboration into an small, core calculus of pure computations. We hope that a unifying framework like ours will make it easier for the programmers to adopt and benefit from the great many indexing structures from the literature.

2 INDEXED EFFECTS IN F^* , BY EXAMPLE

This section introduces indexed effects in F^* progressively, starting with a simple, non-indexed state monad and working our way eventually to a graded, 2-state Hoare monad for functional correctness proofs for stateful programs. We emphasize two points:

- By carefully designing the indexing structure on a monadic effect, it is possible to reason about programs in an abstraction well-suited to the reasoning task at hand.
- Regardless of the indexing structure, e.g., whether no indices are used at all, or if the indices are drawn from some rich logic, our type-and-effect directed elaboration helps in hiding the complexity of the underlying semantic models of an effect from a programmer, providing, in addition to syntactic elaboration, features such as automated subsumption and coercion between effects.

2.1 Background on F^* and a non-indexed effect for state

We start with a review of F^* and show how to define a simple effect based on an ordinary state monad.

³We submit all the examples presented in the paper as anonymous supplementary material.

F* is a program verifier and a proof assistant based on a dependent type theory with a countable hierarchy of predicative universes (like Coq or Agda). Proofs in F* are partially automated using the Z3 SMT solver [de Moura and Bjørner 2008], although it also has a 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. Rather than focusing on purely functional programming, F* has been used extensively to build security-critical, high-performance, low-level software in several embedded DSLs. The resulting code has been deployed in a variety of settings, including the Windows kernel, the Linux kernel, the Microsoft Azure cloud, the Firefox web browser, and several other applications where high-assurance effectful programs are necessary. In service of these scenarios, an integral part of F* is its ability to be extended with user-defined effects. To date, effects in F* have been tied to Dijkstra monads [Ahman et al. 2017; Maillard et al. 2019; Swamy et al. 2013]—no longer, as we will soon see.

Basic syntax. F* syntax is roughly modeled on OCaml (`val`, `let`, `match`, etc.). Binding occurrences `b` 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 `λb1 ... bn → t` introduces a lambda abstraction (metavariable `t` ranges over both types and terms), whereas `b1 → ... → bn → C` is the shape of a curried function type with *computation type* `C` (more about them shortly). Refinement types are written `b{t}`, e.g., the type `x:int{x ≥ 0}` represents natural numbers. We define `squash t` as the unit refinement `λ_.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. E.g., the type `#a:Type → #m:nat → #n:nat → vec a m → vec a n → vec a (m + n)` represents the append function on vectors, 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, writing the type of `append` as just `vec a m → vec a n → vec a (m + n)`. We also omit universe annotations.

Returning to the computation types `C`, F* distinguishes computations from values in a manner similar, though not identical, to Levy’s (2004) Call-By-Push-Value calculus. Computation types include `Tot t` (`x:t1 → t2` is a shorthand for `x:t1 → Tot t2`) for pure, total computations. Another built-in computation type is `Lemma (requires p) (ensures q)`, which is the type of a computation which when executed in a context validating `p` terminates in a context validating `q`, i.e., it can be seen as sugar for `squash p → squash q`. When `p` is trivial, we simply write `Lemma (ensures q)` or `Lemma q`.

F* also allows users to define new computation types, however, to date, every user-defined computation type was required to be a Dijkstra monad of predicate transformers, either axiomatized [Swamy et al. 2016] or derived using a CPS transformation of programs in a sub-language of effect definitions [Ahman et al. 2017]. Defining even a simple non-indexed state monad as computation type was not possible, until now.

A simple state monad and its corresponding effect. Defining a state monad in F* is easy, just as in many functional languages.

```
let st (a:Type) (s:Type) = s → a & s (* & is the tuple type constructor *)
let return (x:a) s : st a s = λs → x, s
let bind (f:st a s) (g:a → st b s) : st b s = λs → let x, s' = f s in g x s'
let get () : st s s = λs → s, s
let put (x:s) : st unit s = λ_ → (), x
```

One can, of course, write programs like this `bind (get()) (λs → put (s + 1))` to increment the state, and F* will infer the type `st unit int` for it. However, this style quickly becomes cumbersome. While many languages offer a `do`-notation for monads (F* does too) and in the case of ordinary

monads such as this one, the do-notation is adequate, indexed effects provide an alternative which scales also to indexed monads. With indexed effects, F^* allows users to define a new effect, using the notation below:

```
effect { ST (a:Type) (s:Type) with { repr = st; return; bind; get; put } }
```

We use the term “effect” to refer to a computation type constructor. Here, ST is an effect and the definition above introduces a new user-defined computation type, $ST\ a\ s$, whose underlying representation is $st\ a\ s$, supporting the return and bind combinators, and two actions $ST?.get : unit \rightarrow ST\ s\ s$ and $ST?.put : s \rightarrow ST\ unit\ s$ —the notation $ST?.op$ names the operation op in the ST effect declaration.⁴

With this in place, one can write $ST?.put\ (ST?.get()\ +\ 1)$ and have F^* infer the type $ST\ unit\ int$, while elaborating the program internally to the explicitly monadic notation shown earlier. Since computation types can appear only to the right of an arrow, corresponding to a call-by-value evaluation strategy, and by enforcing left-to-right evaluation order, the elaboration into the explicitly monadic notation becomes algorithmic.

For ordinary monads, this may not seem like much. Indeed, what we have here corresponds closely to a type-and-effect elaboration into explicitly monadic notation developed previously for ML-like programs and ordinary monads by Swamy et al. [2011]. A main contribution of this paper is to show how this basic idea can be generalized to work in a dependently typed setting with indexed monads of all flavors.

2.2 Indexed effects, in a nutshell

An effect declaration in F^* allows promoting any indexed monad m into an effect M . Doing so requires:

- (1) A representation type, $m\ t\ \bar{i}$, for an arbitrary arity $|\bar{i}|$.
- (2) A return, whose type is of the form $x:a \rightarrow m\ a\ \bar{p}$, for some \bar{p} .
- (3) A bind, whose type is of the form $m\ a\ \bar{p} \rightarrow (x:a \rightarrow m\ b\ \bar{q}) \rightarrow m\ b\ \bar{r}$, for some $\bar{p}, \bar{q}, \bar{r}$.
- (4) Zero or more actions, \bar{a} where each a_i has a type of the form $\bar{x}_i : t_i \rightarrow m\ s_i\ \bar{p}_i$.
- (5) An optional subsumption combinator, $subcomp, _ : m\ a\ \bar{p}\ \{pre\} \rightarrow m\ a\ \bar{q}$, which allows re-indexing an $m\ a\ \bar{p}$ to an $m\ a\ \bar{q}$, when $pre : prop$ is valid.
- (6) An optional branching combinator, if_then_else whose type has the form $m\ a\ \bar{p} \rightarrow m\ a\ \bar{q} \rightarrow bool \rightarrow Type$, such that for all $x:m\ a\ \bar{p}, y:m\ a\ \bar{q}$, and $b:bool$, $if\ b\ then\ subcomp\ x\ else\ subcomp\ y$ has type $if_then_else\ x\ y\ b$.

Having introduced an effect M based on m , our type-and-effect system infers computation types $M\ a\ \bar{i}$ for programs using the effectful actions $M?.a_i$, and automatically elaborates them into the underlying monadic operations on m , while generating verification conditions to show that the inferred type of a program is compatible with a user-provided annotation, implicitly re-indexing terms as needed using $subcomp$ and if_then_else . The resulting verification conditions can be dispatched in F^* using a variety of techniques, ranging from SMT solving to interactive proofs with tactics.

Composing multiple effects. Further, given two effect declarations M and N with representation types m and n , F^* ’s effect system supports implicitly lifting M -computations to N -computations if the programmer supplies a combinator $lift : _ : m\ a\ \bar{i}\ \{pre\} \rightarrow n\ a\ \bar{j}$.

As we will soon see, indexed effects allow one to design multiple, effect-based domain-specific languages in F^* , and for those languages to be composable, while enjoying the full native syntax of F^* (with let bindings, pattern matching, recursion, etc.), verification condition generation, and proof

⁴Strictly speaking, to reference the actions in the GST effect of §1.2, we should have written $GST?.read$ and $GST?.write$, though we omitted the “GST?.” prefix for simplicity.

295 automation in F^* , with a foundation of trust built within F^* upon a model of effectful computations
 296 as indexed monads.

297

298 2.3 A Hoare monad for functional correctness proofs of stateful programs

299 To prove the correctness of stateful programs, Nanevski et al. [2008] proposed to refine the state
 300 monad $st\ a\ s = s \rightarrow a \ \&\ s$ with predicates drawn from a Hoare logic. In our setting, this can be
 301 encoded like so:

```
302 let hst a s (p:s → prop) (q:s → a → s → prop) = s0:s{p s0} → x:(a & s) { q s0 (fst x) (snd x) }
```

304 The type $hst\ a\ s\ p\ q$ represents a state-passing computation which, when run in an initial state
 305 $s0:s$ validating the precondition $p\ s0$, returns a `result:a` and a final state $s1:s$ that validates the
 306 postcondition $q\ s0\ result\ s1$. One can define the following combinators.

```
307 let return (x:a) : hst a s (λ _ → ⊤) (λ s0 r s1 → s0==s1 ∧ r==x) = λs → x, s
308 let bind (f:hst a s p_f q_f) (g: (x:a → hst b s (p_g x) (q_g x)))
309 : hst b s (λ s0 → p_f s0 ∧ (∀ x s1. q_f s0 x s1 ⇒ p_g x s1)) (λ s0 r s2 → (∃ x s1. q_f s0 x s1 ∧ q_g x s1 r s2))
310 = λs0 → let x, s1 = f s0 in g x s1
```

311

312 This is different from a classic Hoare monad, for a few reasons. First, the postcondition we use
 313 here is a predicate covering over both the initial and final state—a so-called, 2-state postcondition.
 314 Further, a standard Hoare monad as sketched in Nanevski et al. [2008] has a `bind` with a signature
 315 resembling Atkey’s parameterized monad in that the postcondition of the f matches the precondition
 316 of g . However, our `bind` for `hst` is designed so that it is parametric in the pre- and postconditions of
 317 both f and g , making type inference and verification condition generation for `hst` easier. To enable
 318 this, we must also strengthen the precondition of the resulting computation with a requirement that
 319 q_f is stronger than p_g , while, to retain precision, the final postcondition is also strengthened with
 320 both q_f and q_g . Non-standard or not, `hst` can be easily promoted to an effect in F^* , since the effect
 321 mechanism places no restrictions on the indexing structure. But, before promoting `hst` to an effect,
 322 we’ll define some actions, a subsumption rule, and a type for composing branching computations.

323 *Actions for hst.* The `get` and `put` actions are computationally equivalent to their unrefined
 324 counterparts in `st`. In `hst`, we give them precise logical specifications.

```
325 let get () : hst s s (λ _ → ⊤) (λ s0 x s1 → s0 == s1 ∧ x == s1) = λs → s, s
326 let put (x:s) : hst unit s (λ _ → ⊤) (λ _ _ s1 → x == s1) = λ_ → (), x
```

328

329 *Subsumption, or the Hoare rule of consequence.* Hoare logics typically include a rule of consequence,
 330 enabling preconditions to be strengthened and postconditions to be weakened. Our Hoare logic
 331 encoded in `hst` also admits such a rule, which we can encode as a subsumption combinator for
 332 re-indexing `hst`, shown below.

```
333 let subcomp (x:hst a p q {relate_pre_post p p' q q'}) : hst a p' q' = x
334 where relate_pre_post p p' q q' = (∀ s. p' s ⇒ p s) ∧ (∀ s0 x s1. p' s0 ∧ q s0 x s1 ⇒ q' s0 x s1)
```

335

336 *Branching.* Whereas `bind` is required to specify how to sequentially compose computations,
 337 within our framework, it is also possible to specify how to type and compose computations under
 338 branches. In this case, to typecheck `if b then f else g` it suffices to take the join of their types by
 339 simply lifting the conditional to the level of the indices.

```
340 let ite (f:hst a s p_f q_f) (g:hst a s p_g q_g) b = hst a s (if b then p_f else p_g) (if b then q_f else q_g)
```

342 Finally, an effect definition promotes `hst` to `HST`, as shown below.

343


```

344 effect { HST (a:Type) (s:Type) (p:s → prop) (q:s → a → s → prop) with
345   { repr = hst; return; bind; subcomp; if_then_else = ite; get; put }}

```

346 To write and prove stateful programs in HST, one starts by picking a model for mutable memory.
347 To illustrate, we choose just a store that maps natural number memory locations (`loc`) to integers,
348 where `Map.t` is a total map from F^* 's standard library, supporting operations to select (`sel`) and
349 update (`upd`) keys in the map. We build derived actions to read and write locations in memory,
350 giving them precise specification in HST. We do not model dynamic allocation nor typed references,
351 as they are orthogonal—several other memory models in F^* support such features [Ahman et al.
352 2018; Protzenko et al. 2017; Swamy et al. 2020] and are usable with indexed effects.

```

354 let loc = nat      let store = Map.t loc int
355 let read (x:loc) : HST int store (λ _ → T) (λ s0 v s1 → s0 == s1 ∧ v = sel s1 x) = sel (HST?.get ()) x
356 let write (x:loc) (v:int) : HST unit store (λ _ → T) (λ s0 _ s1 → s1 == upd s0 x v)
357   = HST?.put (upd (HST?.get()) x v)

```

358 Putting several features together, we implement and prove programs like so, where we tag the
359 pre- and postcondition with the (semantically irrelevant) keywords `requires` and `ensures` just to
360 improve readability:

```

361 let mod_or_sqr (b:bool) (x y:loc) : HST unit store
362   (requires λ s → sel s x > 0)
363   (ensures λ s0 _ s1 → ∃v. (if b then v ≤ sel s0 x else v ≥ sel s0 x) ∧ s1 == upd s0 y v)
364   = if b then write y (read y % read x) else write y (read x * read x)
365

```

366 For even this simple program, the type-and-effect based elaboration is a significant benefit. First,
367 in each branch of the condition, we have imperative code developed directly in an applicative
368 notation, rather than requiring it to be explicitly monadic. Next, although each branch has a different
369 type, the `if_then_else` combinator provides a form of dependent pattern matching, automatically
370 giving the conditional a type that depends on the branch condition `b`. Finally, when the user
371 annotates a specification for a function, the system automatically applies the rule of consequence,
372 building a verification condition, which, in this case, is automatically discharged by SMT.

373 Without the effect-based elaboration, one could try to write a program directly in the `hst` monad.
374 It would look something like this (where `read_hst` and `write_hst` are the `hst` analogs of `read` and
375 `write`), where one essentially has to build a Hoare-style derivation by hand. Even if the system is
376 able to infer all the missing implicit arguments (shown as underscores), which it cannot, in this
377 case, this style is verbose and obscures the program beyond recognition.

```

378 let mod_or_sqr (b:bool) (x:loc) (y:loc)
379   : hst unit store (λ s → sel s x > 0)
380   (λ s0 _ s1 → ∃v. (if b then v ≤ sel s0 x else v ≥ sel s0 x) ∧ s1 == upd s0 y v)
381   = subcomp _ _ _ _ _
382     (if b then subcomp _ _ _ _ _
383       (bind _ _ _ _ _ (read_hst y) (λ v0 →
384         bind _ _ _ _ _ (read_hst x) (λ v1 →
385           bind _ _ _ _ _ (lift_pure_hst _ _ (v0 % v1)) (λ i →
386             write_hst y i))))
387       else subcomp _ _ _ _ _
388         (bind _ _ _ _ _ (read_hst x) (λ v0 →
389           bind _ _ _ _ _ (read_hst x) (λ v1 →
390             write_hst y (v0 * v1))))))
391

```

393 *Sub-effects.* If you look closely at the elaboration above, you may wonder about `lift_pure_hst`
 394 in the `then`-branch. What’s happened here is that our system has automatically lifted a pure
 395 computation with a non-trivial precondition into the `hst` effect. We explain this in greater detail
 396 in §3.4, summarizing the main idea here. At the core of F^* , pure computations with non-trivial
 397 preconditions are typed in a Dijkstra monad `PURE` `a wp`, a type for *conditionally pure* computations
 398 which when evaluated in a context validating `wp p : prop`, for any `p : a → prop`, return a `v : a` vali-
 399 dating `p v`. In this case, we have a term `v0%v1` which, due to a possible division by zero, has type
 400 `PURE int (λp → v1 ≠ 0 ∧ (v1 ≠ 0 ⇒ p (v0 % v1)))`, indicating that it is only safe run the term in context
 401 where `v1 ≠ 0` is valid. To incorporate such conditionally pure computations within HST, one can
 402 specify that `PURE` is a *sub-effect* of HST, as shown below.

```
403 let pure a (wp : (a → prop) → prop) = unit → PURE a wp
404 let lift_pure_hst (f : pure a wp) : hst a
405   (requires λ_ → wp (λ _ → T))
406   (ensures λs0 v s1 → s0 == s1 ∧ ¬(wp (λ y → ¬(y == v))))
407   = λs → f(), s
408 sub_effect PURE ~>HST = lift_pure_hst
409
```

410 Our system maintains a directed acyclic graph of sub-effects, ensuring that two effects are related
 411 by sub-effecting in at most one way. During elaboration, this allows us to unambiguously lift one
 412 effect to another, while relating the indexing abstractions appropriately. In this case, `lift_pure_hst`
 413 interprets the predicate transformer index `wp` as a Hoare-style precondition (applying it to a trivial
 414 postcondition), and as a postcondition, relying on a double-negation transformation of the `wp`.

415 From even this simple example, we hope to illustrate that due to the prohibitive syntactic
 416 overhead, novel indexed monad constructions, despite providing very useful abstractions for
 417 program reasoning, are difficult to adopt in practice. Instead, with our effect elaboration system,
 418 one can freely explore the design space of indexed monads to build suitable abstractions applicable
 419 to programs that humans can write, understand, and prove correct, with good automation. As an
 420 instance of such an exploration, we present next a novel refinement of HST, indexing it additionally
 421 with a non-trivial monoid to track read and write effects and information flows, while still benefiting
 422 from automated elaboration.

423 2.4 Refining HST with information flow control

424 While the Hoare logic encoded in the HST effect is expressive enough for functional correctness
 425 proofs, the specifications it permits are relatively unstructured—pre- and postconditions are just
 426 predicates on the entire store—and are limited to properties of a single execution of a program. In
 427 this section, we present HIFC, a refinement of HST, based on a graded Hoare monad for state, where
 428 computation types of the form `HIFC a reads writes flows pre post` constrain the set of memory
 429 locations read and written, and the dependences among them, in addition to the Hoare-style pre-
 430 and postconditions. We summarize our construction here, with the full details in the appendix.

431 To define the effect HIFC, we’ll start with a indexed monad representation, `hifc`, refining `hst`.

```
432 let label = set loc    let flow = label & label    let flows = list flow
433 let hifc a (r w:label) (fs:flows) p q = f:hst a store p q { writes f w ∧ reads f r ∧ respects f fs }
```

434 Interpreting the write index, `writes f w`, states that all locations not in `w` are unchanged when
 435 running `f` in any state. The reads predicate involves a relational interpretation, similar to [Benton
 436 et al. \[2006\]](#), stating that runs of `f` on stores that differ only on unread locations are equivalent. The
 437 respects relation is the main statement of noninterference, also stated relationally—information
 438 flows from `l` to `m` only if there exists some $(src, dest) \in flows$ such that $l \in src$ and $m \in dest$.

441

Next, we define `return`, to lift pure terms into the `hifc` abstraction; and `bind` to show that the abstraction of `hifc` is stable under sequential composition. The proof of the correctness of `bind` is non-trivial and requires about a few hundred lines of auxiliary lemmas in F^* , but this proof is done once and for all. We just show the signatures.

```

442 val return (x:a) : hifc a {} {} [] (λ _ → T) (λ s0 r s1 → s0 == s1 ∧ r == x)
443
444 val bind (f:hifc a r0 w0 fs0 p q) (g: (x:a → hifc b r1 w1 fs1 (r x) (s x)))
445   : hifc b (r0 ∪ r1) (w0 ∪ w1) (fs0 @ add_source r0 (({} , w1)::fs1))
446     (λ s0 → p s0 ∧ (∀ x s1. q s0 x s1 ∧ modifies w0 s0 s1 ⇒ r x s1))
447     (λ s0 r s2 → (∃ x s1. (q s0 x s1 ∧ modifies w0 s0 s1) ∧ (s x s1 r s2 ∧ modifies w1 s1 s2)))
448
449 where add_source r fs = List.map (λ (src, sink) → (src ∪ r, sink)) fs    and @ is list concatenation

```

The type of `bind f g` has several interesting elements. `bind f g` reads (and writes) the union of the read (and write) sets of `f` and `g`. More subtly, the flows of `bind f g` are the flows of `f (fs0)`, together with the flows of `g ({}, w1)::fs1` augmented with flows from the locations read by `f (r0)`, since `g`'s argument is tainted by `f`'s reads. This way of computing flows is more precise than in prior monadic IFC systems, which usually consider that all locations read by `f` can flow to all locations written by `g`. Finally, showing the use of the write index for framing, both the pre- and postcondition exploit the invariant that `f` does not modify locations outside `w0` and `g` outside `w1` (where the predicate `modifies w s0 s1` states that `s0` and `s1` agree on all locations outside `w`). As such, the write index encodes a form of dynamic framing [Kassios 2006].

We can show that the triple of additional indices of HIFC, (reads, writes, flows) form a monoid (under a suitable equivalence relation) whose unit is `{}, {}, []` and whose elements can be composed with \oplus (shown below), which is the indexing pattern of a graded monad, used on `return` and `bind`.

$$(r_0, w_0, f_0) \oplus (r_1, w_1, f_1) = r_0 \cup r_1, w_0 \cup w_1, f_0 @ \text{add_source } r_0 ((\{\}, w_1)::f_1)$$

Packaging `hifc` as an effect HIFC, together with a subsumption relation that allows widening the reads, writes and flows sets together with the Hoare rule of consequence; a branching combinator; and actions to read and write individual locations, allows us to write and prove effectful programs for both correctness and security, by reasoning only about their indices. For example, `write l1(read l0)` is inferred to have type $\text{HIFC unit } \{l_0\} \{l_1\} [\{l_0\}, \{l_1\}] (\lambda_ \rightarrow T) (\lambda s_0_ s_1 \rightarrow \text{sel } s_1 l_1 == \text{sel } s_0 l_0)$.

Refining flows with Hoare reasoning. Monadic label-based information flow is inherently imprecise, since it conflates data and control dependence. To illustrate, consider `read h; write l (read l + 1)` which has the type $\text{HIFC unit } \{h, l\} \{l\} [\{h\}, \{l\}] (\lambda_ \rightarrow T) (\lambda s_0_ s_1 \rightarrow \text{sel } s_1 l = \text{sel } s_0 l + 1)$, suggesting that it leaks information from `h` to `l`, when in reality no such flow exists since the `read h` is redundant. However, HIFC's pre- and postconditions can be used to recover precision. The re-indexing coercion, `refine`, allows removing a flow `f` from $\text{HIFC a r w (f::fs) p q}$ when the Hoare specifications `p` and `q` allow proving that the `f`-flow is spurious.

```

480 val refine (λ (unit → HIFC a r w (f::fs) p q) { (∀ from to v. from ∈ fst f ∧ to ∈ snd f ∧ from ≠ to ⇒
481   (∀ s0 x x' s1 s1'. (p s0 ∧ p (upd s0 from v) ∧ q s0 x s1 ∧ q (upd s0 from v) x' s1' ∧
482     modifies w s0 s1 ∧ modifies w (upd s0 from v) s1') ⇒
483     sel s1 to == sel s1' to) }) : unit → HIFC a r w fs p q

```

Using `refine` (which the programmer must explicitly apply), we can revise the type of our example term to $\text{HIFC unit } \{h, l\} \{l\} [] \dots$, removing the spurious flow.

We have seen how in the type of `bind` the Hoare specifications are improved using the write index by internalizing framing. Conversely, with `refine`, Hoare specifications also improve the precision of the information flow labels, illustrating the useful interplay between the two indexing

540	constant	$T ::= \text{unit} \mid \text{Type}_u \mid \text{sum} \mid \text{inl} \mid \text{inr}$
541	term	$e, t ::= x \mid T \mid x:t_1\{t_2\} \mid x:t \rightarrow C \mid \lambda x:t. e \mid e_1 e_2 \mid \text{let } x = e_1 \text{ in } e_2$
542		$\mid \text{case}_C(e \text{ as } y) x.e_1 x.e_2 \mid \text{reify } e \mid F.\text{reflect } e$
543	computation type	$C ::= \text{Tot } t \mid F t \bar{e}$ effect label $M ::= \text{Tot} \mid F$
544	effect definition	$D ::= F (a : \text{Type}) \overline{(x : t)} \{e_{repr}; e_{return}; e_{bind}; e_{subcomp}\}$
545	lift definition	$L ::= \text{lift}_{M_1}^{M_2} = e$ signature $S ::= \cdot \mid D, S \mid L, S$
546	type environment	$\Gamma ::= \cdot \mid x : t, \Gamma$ typing context $\Delta ::= S; \Gamma$
547		
548		

Fig. 2. Syntax of IMF*

study these enhancements in the future, grateful to no longer have to consider their interaction with Dijkstra monads.

3.2 IMF*: A surface language with user-defined indexed effects

IMF* models a surface language with support for user-defined indexed effects. Figure 2 shows the syntax. IMF* adds effectful constructs to TotalF*. These include computation types $F t \bar{e}$ (where F is the effect, t is the return type, and \bar{e} are the effect indices) in C , let bindings, and reify and reflect coercions between the computation types and their underlying representations.

An effect definition D defines an indexed effect F with indices types $\overline{x : t}$ and combinators e_{repr} , e_{return} , e_{bind} , and $e_{subcomp}$, while $\text{lift}_{M_1}^{M_2}$ defines a combinator to lift M_1 computations to M_2 (effect M ranges over Tot and F). Our implementation allows for specifying an optional custom effect combinator for combining the branches of case—we discuss it in §3.5.

IMF* inherits the rest from TotalF*. Thus, the monadic structure of the terms is implicit in the surface syntax, and is elaborated into explicit binds and lifts by the typing judgment.

The main typechecking judgment in IMF* has the form $\Delta \vdash e : C \rightsquigarrow e'$ stating that under a typing context Δ , expression e has computation type C and elaborates to expression e' in TotalF*. When the elaborated term is not significant, we just write $\Delta \vdash e : C$.

To illustrate the typing rules, we use the graded state monad from Section 1, partly reproduced here, as a running example. One difference, to illustrate the use of `reflect`, is that rather than including the read and write actions as part of the effect definition, we show a desugared form where we use `GST.reflect` to promote a `gst a t` to a `GST a t` computation type, separately from the effect definition.

```

574 let gst (a:Type) (t:tag) = s0:state → r:(a & state) { t=R ⇒ s0 == snd r }
575 let return a (x:a) : gst a R = λs → x,s
576 let bind a b t0 t1 (f:gst a t0) (g: a → gst b t1) : gst b (t0 ⊕ t1) = λs0 → let x, s1 = f s0 in g x s1
577 effect { GST (a:Type) (t:tag) with { repr=gst; return; bind } }
578 let read () : GST state R = GST.reflect (λ s → s,s)
579 let write s : GST unit RW = GST.reflect (λ _ → (), s)

```

Typechecking effect definitions and lifts. While IMF* does not impose any constraints on the layering or indexing structure of the effects, the types of the combinators in an effect definition D are constrained to have specific shapes, as described in §1.2. Whereas previously we left the additional index arguments in these combinators implicit, here, to be clearer, we make them explicit. We write $F.repr$, $F.bind$ etc. to denote e_{repr} , e_{bind} etc. for the effect F in an ambient signature S .

An effect definition for F is typechecked as follows. $F.repr$ has a function type with argument types matching the effect signature $F (a : \text{Type}) \overline{x : t}$. $F.return$ and $F.bind$ have monad-like shapes

589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637

$$\begin{array}{c}
\text{T-VAR} \\
\frac{x : t \in \Gamma}{\Delta \vdash x : t \rightsquigarrow x} \\
\\
\text{T-LET} \\
\frac{\Delta \vdash e_1 : C_1 \rightsquigarrow e'_1 \quad \Delta \vdash \lambda x. e_2 : x : C_1^t \rightarrow C_2 \rightsquigarrow e'_2 \quad x \notin FV(C_2^t)}{\Delta, f : \hat{C}_1, g : x : C_1^t \rightarrow \hat{C}_2 \vdash C.\text{bind}(\text{lift}_{C_1}^{\hat{C}} f)(\lambda x. \text{lift}_{C_2}^{\hat{C}}(g x)) : \hat{C} \rightsquigarrow e'} \\
\Delta \vdash \text{let } x = e_1 \text{ in } e_2 : C \rightsquigarrow e'[e'_1/f][e'_2/g] \\
\\
\text{T-REIFY} \quad \text{T-REFLECT} \quad \text{T-SUB} \\
\frac{\Delta \vdash e : C \rightsquigarrow e'}{\Delta \vdash \text{reify } e : \hat{C} \rightsquigarrow e'} \quad \frac{\Delta \vdash e : \hat{C} \rightsquigarrow e'}{\Delta \vdash F.\text{reflect } e : C \rightsquigarrow e'} \quad \frac{\Delta \vdash e : C' \rightsquigarrow e' \quad \Delta \vdash C' <: C}{\Delta \vdash e : C \rightsquigarrow e'} \\
\\
\text{T-CASE} \\
\frac{\Delta \vdash e : \text{sum } t_1 t_2 \rightsquigarrow e' \quad \Delta, y : \text{sum } t_1 t_2 \vdash C : \text{Type} \rightsquigarrow t' \quad i \in 1, 2 \quad T = \text{inl if } i = 1, \text{inr o/w}}{\Delta, x : t_i \vdash e_i : C_i[T x/y] \rightsquigarrow e'_i \quad \Delta, x : t_i, f_i : \hat{C}_i[T x/y] \vdash \text{lift}_{C_i}^{\hat{C}} f_i : \hat{C}[T x/y] \rightsquigarrow e''_i} \\
\Delta \vdash \text{case}_C(e \text{ as } y) x.e_1 x.e_2 : C[e/y] \rightsquigarrow \text{case}_{e'}(e' \text{ as } y) x.e''_1[e'_1/f_1] x.e''_2[e'_2/f_2] \\
\\
\text{C-M} \quad \text{SC-M} \quad \text{S-SUB} \\
\frac{\Delta \vdash \hat{C} : \text{Type} \rightsquigarrow t'}{\Delta \vdash C : \text{Type} \rightsquigarrow \text{Tot } t'} \quad \frac{\Delta \vdash C_1 : \text{Type} \rightsquigarrow _ \quad \Delta \vdash \hat{C} <: \hat{C}_1}{\Delta \vdash C <: C_1} \quad \frac{\Delta \vdash t_1 : \text{Type} \rightsquigarrow t'_1 \quad \Delta \vdash t_2 : \text{Type} \rightsquigarrow t'_2}{\Delta \vdash t_1 <: t_2}
\end{array}$$

Fig. 3. IMF* typing judgments (primed symbols are TotalF* syntax)

with unconstrained $\overline{x : t}$ binders that may appear in the index terms $\overline{e_f}$, $\overline{e_g}$, and \overline{e} . We return to the $F.\text{subcomp}$ combinator and the related `if_then_else` combinator later.

$$\begin{array}{l}
S; \cdot \vdash F.\text{repr} \quad : \quad a : \text{Type} \rightarrow \overline{x : t} \rightarrow \text{Type} \\
S; \cdot \vdash F.\text{return} \quad : \quad a : \text{Type} \rightarrow v : a \rightarrow \overline{x : t} \rightarrow F.\text{repr } a \overline{e} \\
S; \cdot \vdash F.\text{bind} \quad : \quad a b : \text{Type} \rightarrow \overline{x : t} \rightarrow F.\text{repr } a \overline{e_f} \rightarrow (x : a \rightarrow F.\text{repr } b \overline{e_g}) \rightarrow F.\text{repr } b \overline{e}
\end{array}$$

Turning to our example, we have $\text{GST.repr} = \text{gst}$, with one effect index $t:\text{tag}$. $\text{GST.return} = \text{return}$, and in this case we have no additional index arguments. $\text{GST.bind} = \text{bind}$, with the t_0 and t_1 arguments to `bind` being the $\overline{x : t}$ binders. When applying these combinators, our implementation relies on F*'s existing higher-order unifier to compute instantiations of the a, b type arguments, and the $\overline{x : t}$ arguments.

Finally, an expression e defining a lift from F to F' is typechecked as a coercion from $F.\text{repr}$ to $F'.\text{repr}$, i.e., $S; \cdot \vdash e : a : \text{Type} \rightarrow \overline{x : t} \rightarrow F.\text{repr } a \overline{e_f} \rightarrow F'.\text{repr } a \overline{e}$. Every user-defined effect in IMF* gets an automatic lift from Tot : $\text{lift}_{\text{Tot}}^F = F.\text{return}$. We discuss checks we impose on the lifts collectively to ensure coherence in §3.5.

3.3 Type-and-effect directed elaboration

The typing rules from Figure 3 elaborate the implicitly monadic IMF* terms to TotalF* by inserting binds and lifts. The placement of these combinators are purely syntax-directed, since computation types can only appear immediately to the right of an arrow, and because we enforce left-to-right evaluation order. However, applying the `bind` and `lift` combinators requires inference of the effect indices and the combinator arguments (e.g., for the $\overline{x : t}$ binders in the combinator types). For this, the system includes a declarative, non-coercive subtyping rule, and implicit arguments in all the rules are chosen nondeterministically. In §3.5, we discuss how this nondeterminism is resolved using F*'s higher-order unifier and annotation-driven subtyping algorithm.

We adopt some notational conventions. We use C^t to project the return type from a C . We hide the context Δ from the lookup notations when it is clear from the context. \hat{C} is the underlying representation of C , defined as t when $C = \text{Tot } t$ and $F.\text{repr } t \bar{e}$ when $C = F t \bar{e}$. We also elide the type t from $\lambda x:t. e$ when it is clear from the context. $C.\text{bind}$ looks up the bind combinator for the effect of C (for $C = \text{Tot } _$, bind desugars to function application). Finally, we write $\text{lift}_{C_1}^{C_2}$ to mean the lift combinator $\text{lift}_{M_1}^{M_2}$ in S , where M_i is the effect of C_i .

Rule T-VAR is the standard variable typing rule, elaborated as is to TotalF*. Rule T-LET is the let-binding rule. Whereas EMF* had explicit monadic binds and lifts in the syntax, in IMF*, monadic elaboration is type-directed and more accurately describes F*'s implementation. The rule first typechecks $e_1:C_1$ and $e_2:C_2$. Since C_1 and C_2 could have different effects, the rule lifts them to a common computation type C , and binds the resulting C 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 $f : \hat{C}_1^t$ and $g : x:C_1^t \rightarrow \hat{C}_2$, applies the lift combinators to f and g , and then applies the resulting computations to $C.\text{bind}$. 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 .

T-REIFY and T-REFLECT move back-and-forth between a computation type and its representation. Interestingly, the elaboration of $\text{reify } e$ (resp. $M.\text{reflect } e$) is just the elaboration of e ; reify and reflect are just identity coercions in IMF* 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 now return to the GST increment example from Section 1 and show the typing rules at work. To elaborate $\text{let } x = \text{read } () \text{ in write } x+1$, the rule T-LET first typechecks $\text{read } ()$:GST state R and elaborates it to $(\lambda s \rightarrow s,s)$ ⁵ (the elaboration uses the rules for application and lambda forms which we present in the supplementary material; the rules are straightforward and descend into their subterms as expected). Note that the definition of read uses reflect , which is an identity in TotalF*. Next, the rule typechecks $\lambda x \rightarrow \text{write } x+1$:state \rightarrow GST unit RW and elaborates it to $(\lambda x _ \rightarrow (), x+1)$. Since the two effect labels are the same, and GST.bind is already a TotalF* term, the final elaborated term is $\text{GST.bind } (\lambda s \rightarrow s,s) (\lambda x _ \rightarrow (), x+1)$.

Rule T-CASE is similar to T-LET. 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 C 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.

The rule T-SUB applies subtyping on computations. 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, with any preconditions arising as proof obligations expressed within TotalF*'s \vDash entailment relation (dispatched in practice to SMT or to user tactics). Since the rule does not automatically lift C , it does not need to be coercive. Similarly, rule S-SUB lifts TotalF*'s subtyping rule for use with IMF*'s value types.

Our main theorem states that the IMF* translation to TotalF* is type-preserving.

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

Here, $[[\Gamma]]_S$ is the pointwise translation of the typing environment, and $[[\Gamma]]_S \vdash e' : t'$ is the typing judgment in TotalF*. Using the theorem, a typing derivation in IMF* can be soundly interpreted in TotalF*. Ahman et al. [2017] prove EMF* normalizing, type-preserving, and a consistency

⁵The elaborated term is actually $(\lambda () s \rightarrow s,s) ()$, we eliminate the application for clarity.

property for its refinement logic—these results also apply to its TotalF^{*} fragment. 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 $S; \Gamma \vdash t : \text{Type} \rightsquigarrow t'$ then $S; \Gamma \vdash t_1 : \text{Type} \rightsquigarrow t'_1$ and $[\Gamma]_S \vdash t' <: t'_1$.

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

The essence of effect abstraction: Admissibility of using $F.\text{subcomp}$ 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 $F.\text{subcomp}$ effect combinator.

The $F.\text{subcomp}$ combinator is typechecked, once-and-for-all as part of the effect definition, as follows:

$$S; \cdot \vdash F.\text{subcomp} : a : \text{Type} \rightarrow \overline{x} : t \rightarrow f : F.\text{repr } a \overline{e}\{t_1\} \rightarrow F.\text{repr } a \overline{e}_1$$

and

$$F.\text{subcomp} = \lambda a \overline{x} f. f$$

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 $F.\text{subcomp}$ is required to be an identity function. This is because subtyping in F^{*} is intentionally non-coercive, so that the application of subtyping does not disturb equality. To prove $F t \overline{e} <: F t \overline{e}_1$, we check that:

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

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

LEMMA 3.3 (SOUNDNESS OF e_{subcomp}). If $\Delta \vdash F t \overline{e} : \text{Type}$, $\Delta \vdash F t \overline{e}_1 : \text{Type}$, and $\Delta, f : F.\text{repr } t \overline{e} \vdash F.\text{subcomp } f : F.\text{repr } t \overline{e}_1$, then $\Delta \vdash F t \overline{e} <: F t \overline{e}_1$.

The proof of the lemma unfolds $F.\text{subcomp}$ to prove the subtyping of the representations, after which an application of SC-M gives us the conclusion. Since it is admissible, we preserve the effect abstractions and use the $F.\text{subcomp}$ combinator to check subsumption, rather than implementing SC-M as is. Additionally, this means that when implementing type conversion in IMF^{*}, we do not need to translate types all the way down to TotalF^{*}—Lemma 3.3 assures us that such a translation would always succeed.

3.4 Encoding PURE, F^{*}'s primitive Dijkstra monad

Prior to indexed effects, effectful computation types in F^{*} had a fixed shape $M a w$, where w is an M -specific weakest precondition predicate transformer [Swamy et al. 2016]. The most primitive Dijkstra monad in F^{*} is for conditionally pure computations, written as PURE ($a:\text{Type}$) ($w:\text{wp } a$), where $\text{wp } a$ is the type of a monotonic predicate transformer, transforming an a -predicate into a precondition.

$$\text{let } \text{wp } a = w : (a \rightarrow \text{prop}) \rightarrow \text{prop} \{ \forall p_1 p_2. (\forall x. p_1 x \implies p_2 x) \implies (w p_1 \implies w p_2) \}$$

Rather than taking it as primitive, PURE can be defined as an indexed effect whose representation is $\text{pure } a w$, a form of continuation monad where the $p:(a \rightarrow \text{prop})$ is a “logical continuation” in prop .

$$\text{let } \text{pure } a w = p : (a \rightarrow \text{prop}) \rightarrow \text{squash } (w p) \rightarrow v : a \{ p v \}$$

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

$$\text{let } \text{bind } (f : \text{pure } a w_1) (g : (x : a \rightarrow \text{pure } b (w_2 x))) : \text{pure } b (\lambda p \rightarrow w_1 (\lambda x \rightarrow w_2 x p)) =$$

$$\lambda p \rightarrow \text{let } x = f (\lambda x \rightarrow w_2 x p) () \text{ in } g x p () \text{ (* run } f \text{ with a chosen postcondition, then run } g \text{ with } p \text{ *)}$$

$$\text{let } \text{subcomp } (w_1 w_2 : \text{wp } a) (_ : \text{squash } (\forall p. w_2 p \implies w_1 p)) (f : \text{pure } a w_1) : \text{pure } a w_2 = f$$

$$\text{let } \text{if_then_else } (w_1 w_2 : \text{wp } a) (f : \text{pure } a w_1) (g : \text{pure } a w_2) (b : \text{bool}) = \text{pure } a (\text{if } b \text{ then } w_1 \text{ else } w_2)$$

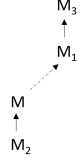
$$\text{effect } \{ \text{PURE } (a:\text{Type}) (w:\text{wp } a) \text{ with } \{ \text{repr} = \text{pure}; \text{return}; \text{bind}; \text{subcomp}; \text{if_then_else} \} \}$$

PURE a w is the type of a pure computation whose result satisfies ρ , provided $w \rho$ holds.

3.5 Implementation of indexed effects

We have implemented 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 $\text{lift}_M^{M_1}$ to the signature, our implementation computes all the new lift edges that it induces via transitive closure. For example, if $\text{lift}_{M_2}^M$ and $\text{lift}_{M_1}^{M_3}$ already exist, this new lift induces a $\text{lift}_{M_2}^{M_3}$ 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. 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.



Algorithmic subtyping. F^* implements a kind of bidirectional type inference algorithm, combined with constraint-based higher-order unification. By propagating programmer-provided annotations through a typing derivation, subtyping is applied only when there is an expected type from the context. Our implementation piggybacks on this infrastructure, relating computation types with the `subcomp` combinators whenever needed.

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

$$S; \cdot \vdash F.\text{if_then_else} : a : \text{Type} \rightarrow \bar{x} : t \rightarrow f : F.\text{repr } a \ \bar{e}_{\text{then}} \rightarrow g : F.\text{repr } a \ \bar{e}_{\text{else}} \rightarrow b : \text{bool} \rightarrow \text{Type}$$

and

$$F.\text{if_then_else} = \lambda a \ \bar{x} \ f \ g \ b. F.\text{repr } a \ \overline{e_{\text{composed}}}$$

F^* ensures the soundness of the combinator by checking that under the assumption b , the type of f is a subtype (as per $F.\text{subcomp}$) of the composed type, similarly for g under the corresponding assumption $\text{not } b$. 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 $F \ t_1 \ \bar{e}_1$ and we want to apply the $\text{lift}_F^{F'}$ combinator, where $\text{lift}_F^{F'} = e$ such that:

$$S; \cdot \vdash e : a : \text{Type} \rightarrow \bar{x} : t \rightarrow F.\text{repr } a \ \bar{e}_f \rightarrow F'.\text{repr } a \ \bar{e}$$

To apply this combinator, we create fresh unification variables for the binders a and \bar{x} , and substitute them with the unification variables in $F.\text{repr } a \ \bar{e}_f$ and $F'.\text{repr } a \ \bar{e}$, without unfolding the e_{repr} . We then unify t_1 with the unification variable for a , \bar{e}_1 with substituted \bar{e}_f , and return (substituted) $F' \ a \ \bar{e}$ as the lifted computation type. This allow us to compute instantiations of the combinators without reifying $F \ t_1 \ \bar{e}_1$ or reflecting the result type of lift. We follow this recipe for all the effect combinators.

In our GST state increment example, to bind the two computation types GST int R (for read) and GST unit RW (for write), using the `bind` combinator:

```
let bind a b t0 t1 (f:gst a t0) (g: a → gst b t1) : gst b (t0 ⊕ t1) = ...
```

we create fresh unification variables $?u_a, ?u_b, ?u_{t_0}, ?u_{t_1}$ for the a, b, t_0, t_1 arguments. We then unify the indices of the f argument, i.e. $?u_a$ and $?u_{t_0}$, with the indices of the first computation type `int` and `R`. Similarly, for the g argument, $?u_b$ and $?u_{t_1}$ are unified with `unit` and `RW`. Finally,

the returned computation type is $\text{GST } ?u_b (?u_{t_0} \oplus ?u_{t_1})$, which after solving for unification variables becomes GST unit RW .

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

4 DIJKSTRA MONADS AND ALGEBRAIC EFFECTS

Maillard et al. [2019] present Dijkstra monads as a monad-like structure M indexed by a separate monad of specifications W , forming types of the shape $M \text{ a } w$, where $w : W \text{ a}$. In this section, we present a refined instance of such a construction, a *graded* Dijkstra monad, integrated within a library of 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]. We show that algebraic effects and handlers can be encoded generically using dependent types in F^* , and exposed to programmers as an 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.

4.1 A graded monad for algebraic effects

Our starting point is a canonical free monad representation $\text{tree}_0 \text{ a}$ of computations with generic actions producing a -typed results. We include stateful operations (Read and Write) and exceptions (Raise), but other operations can be easily added.⁶

```

type op = | Read | Write | Raise
let op_inp o = match o with | Read → unit | Write → state | Raise → exn
let op_out o = match o with | Read → state | Write → unit | Raise → empty
type tree_0 (a : Type) = | Return : a → tree_0 a
                        | Op : op:op → i:(op_inp op) → k:(op_out op → tree_0 a) → tree_0 a

```

The type tree_0 contains all possible combinations of the operations. To limit the operations that may appear in a computation, our representation type tree is indexed by a set of operations that over-approximates the operations in the computation. Specifically, a computation abides by a set of labels labs if its operations are a subset of labs . We use a list for the index, but only interpret it via membership, so order and multiplicity are irrelevant. This makes tree a graded monad, where the monoid operation is the set union.

```

let ops = list op      let tree (a:Type) (labs:ops) = c:(tree_0 a){abides labs c}
where let rec abides (labs:ops) (c : tree_0 a) : prop = match c with
      | Return _ →  $\top$  | Op a i k →  $a \in \text{labs} \wedge (\forall o. \text{abides labs } (k \ o))$ 

```

⁶Our implementation in the supplementary material contains an additional uninterpreted $\text{Other} : \text{int} \rightarrow \text{op}$, and never relies on knowing the full set of operations.

834 *Packaging the refined free monad as an effect.* Elevating tree to an indexed effect is straightforward,
 835 only requiring to define the basic combinators, with most of the heavy lifting done by F*’s SMT
 836 backend. We also provide a subsumption rule to grow the labels where needed, and also reorder
 837 and deduplicate them, since they are essentially sets.

```
838 let return a (x:a) : tree a [] = Return x
839 let bind a b labs1 labs2 (c : tree a labs1) (f : a → tree b labs2) : tree b (labs1 ∪ labs2) = (* elided *)
840 let subcomp a labs1 labs2 (c:squash (labs1 ⊆ labs2)) (f : tree a labs1) : tree a labs2 = f
841 let cond (a:Type) (labs1 labs2 : ops) (f:tree a labs1) (g:tree a labs2) (p: bool) = tree a (labs1 ∪ labs2)
842 effect { Alg (a:Type) (labs:ops) with { repr = tree; return; bind; subcomp; if_then_else = cond }}
843
```

845 4.2 Operations and their handlers, with reflect and reify

846 To add operations to our new effect Alg, the reflect operator is useful, as seen in the generic action
 847 geneff below, which uses reflect to promote a tree to an Alg. Specific instances of operations can
 848 be defined easily using geneff, where for raise we take an extra step of matching on its (empty)
 849 result to make it polymorphic.

```
850
851 let geneff (o : op) (i : op_inp o) : Alg (op_out o) [o] = Alg.reflect (Op o i Return)
852 let get () : Alg state [Read] = geneff Read () let put (s:state) : Alg unit [Write] = geneff Write s
853 let raise (e:exn) : Alg α [Raise] = match geneff Raise e with (* empty match *)
```

854 With this, we can already write simple programs in direct style, while the system infers refined
 855 types and elaborates programs to their tree representation.

```
856 exception Failure of string
857
858 let add_st x : Alg int [Read; Raise] = let s = get () in if s < 0 then raise (Failure "error") else x+s
```

859 When defining effect handlers, one needs access to the tree representation. For instance, the
 860 handle_tree combinator allows all the labs₀ operations in c to be handled by h, which in turn may
 861 leave the labs₁ operations to be handled, with v the continuation of c’s return.

```
862
863 let handler_tree_op o b labs = op_inp o → (op_out o → tree b labs) → tree b labs
864 let handler_tree labs0 b labs1 = o:op{o ∈ labs0} → handler_tree_op o b labs1
865 let rec handle_tree (c : tree a labs0) (v : a → tree b labs1) (h : handler_tree labs0 b labs1)
866   : tree b labs1 = match c with | Return x → v x
867   | Op act i k → h act i (λ o → handle_tree (k o) v h)
```

868 However, rather than calling handle_tree directly, which would require client code to work with
 869 tree, we provide the following interface instead, using reify in negative positions to coerce Alg to
 870 tree and reflect to move back.

```
871
872 let handler labs0 b labs1 = o:op{o ∈ labs0} → op_inp o → (op_out o → Alg b labs1) → Alg b labs1
873 let handle_with (f : unit → Alg a labs0) (v : a → Alg b labs1) (h : handler labs0 b labs1) : Alg b labs1
874   = (* elided, essentially a wrapper of handle_tree, translating h into a handler_tree, etc. *)
```

875 This allows us to write handlers in a direct, applicative notation, close to what is offered by
 876 languages specifically designed for algebraic effects.

```
877
878 let defh : handler labs b labs = λ o i k → k (geneff o i) (* essentially rebuilding an Op node *)
879 let catchE (f : unit → Alg a (Raise::labs)) : Alg (option a) labs =
880   handle_with f Some (function Raise → (λ i k → None) | _ → defh)
```

881

882

4.3 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, it is unclear what can be verified, unless equations are added.

Our new effect will refine tree with a stateful WP. We can define a function that takes a computation tree and computes a “default” stateful WP from it. Then, we take the representation `treewp a l wp` to be 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—`st_wp a` is the type of functions transforming postconditions on results and states to preconditions on states.

```

883 type post_t (a:Type) = a → state → prop
884 type st_wp (a:Type) = state → post_t a → prop
885
886 let read_wp : st_wp state = λs_0 p → p s_0 s_0
887 let write_wp : state → st_wp unit = λs _ p → p () s
888 let return_wp (x:a) : st_wp a = λs_0 p → p x s_0
889 let bind_wp (w_1 : st_wp a) (w_2 : a → st_wp b) : st_wp b = λs_0 p → w_1 s_0 (λ y s_1 → w_2 y s_1 p)

```

Next, we interpret Read-Write trees into `st_wp` via `interp_as_wp`. We refine the tree type by both limiting its operations and adding a refinement comparing its WP via (\leq), the strengthening relation on stateful WPs, and defining return, bind, subcomp, cond, get and put for `treewp`, and promote it to the AlgWP effect. AlgWP is proven sound by interpreting it into the PURE effect from §3.4.

```

890 let rec interp_as_wp #a (t : tree a [Read; Write]) : st_wp a = match t with
891   | Return x → return_wp x
892   | Op Read _ k → bind_wp read_wp (λ s → interp_as_wp (k s))
893   | Op Write s k → bind_wp (write_wp s) (λ () → interp_as_wp (k ()))
894
895 type rwops = l:ops{l ⊆ [Read; Write]}
896 let treewp (a : Type) (l:rwops) (w: st_wp a) = t:(tree a l){ w ≤ interp_as_wp t }
897 effect { AlgWP (a:Type) (l:rwops) (w:st_wp a) with { repr = treewp; ...} }
898 let soundness (t : unit → AlgWP a l wp) : s_0:state → PURE (a & state) (wp s_0) = ...

```

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, where AlgPP is simply a pre-/postcondition alias to AlgWP.

```

899 effect AlgPP a l p q = AlgWP a l (λ s_0 post → p s_0 ∧ (∀ x s_1. q x s_1 ⇒ post x s_1))
900 let swap (l_1 l_2 : loc) : AlgPP unit [Write; Read] (requires λ_ → l_1 ≠ l_2)
901   (ensures λ h_0 _ h_1 → h_0[{l_1;l_2}] == h_1[{l_1;l_2}] ∧ h_1.[l_1] == h_0.[l_2] ∧ h_1.[l_2] == h_0.[l_1])
902   = let r = !l_2 in l_2 := !l_1; l_1 := r

```

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.

```

903 val quotient (f : unit → AlgPP a [Read] p q) : AlgPP a [Read] p (λ h_0 x h_1 → q h_0 x h_1 ∧ h_0 = h_1)

```

In summary, this case study has shown that we can develop dependently typed libraries for sophisticated effect disciplines, provide reasoning principles for them in the form of novel, hybrid indexed monads, and package it up as an effect that enables programs and their proofs to developed at a palatable, high-level of abstraction.

5 LAYERED INDEXED EFFECTS FOR MESSAGE FORMATTING IN TLS

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 stacking two effect layers over EverParse [Ramananandro et al. 2019], an existing library in F* for verified low-level binary message parsing and formatting, simplifying different aspects of the programs and proofs in each layer.

*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 Stack effect which models programming with mutable locations on the stack and heap, with explicit memory layout and lifetimes. Stack is a Hoare monad with the following signature:

```
effect Stack (a:Type) (pre:mem → prop) (post:mem → a → mem → prop)
```

Programs in Stack may only allocate on the stack, while reading and writing both the stack and the heap, with pre- and postconditions referring to 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—we aim to simplify these proofs for binary message formatters with domain-specific abstractions built using indexed effects.

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:

```
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) (x y: U32.t) : Stack bool
  (requires λm0 → live m0 output ∧ len == length output) (* memory safety *)
  (ensures λm0 success m1 → modifies output m0 m1 ∧ (* memory safety & framing *)
   (success ⇒ valid_from parse_example m1 output 0)) (* output correctness wrt parser *)
= if len < 8 then false (* output buffer too small *)
  else let off = write_u32 output 0 x in let _ = write_u32 output off y in
      let mem = get () in example_intro mem output 0; true
```

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

981 involves (crucially) proving that the writing of the second field write does not overlap the first one,
 982 through Low* memory framing. These proofs are here implicit but still incur verification cost to
 983 the SMT solver; as the complexity of the structs increases, it has a significant impact on the SMT
 984 proof automation. Moreover, the user also needs to worry about the size of the output buffer being
 985 large enough to store the two integer fields.

986 We describe next how using indexed effects we can reduce the programming and proving
 987 overhead to:

```
988 let write_example (x y : U32.t) : FWrite unit parse_empty parse_example = write_u32 x; write_u32 y
```

990 5.1 The Write effect

992 We define a Write effect, layered over Stack, to abstract the low-level byte layout, memory safety, and
 993 error handling—we will address framing later. An effectful computation $f : \text{Write } a \text{ pbefore pafter}$
 994 returns a value of type a while working on a hidden underlying mutable buffer. Each such computa-
 995 tion requires upon being called that the buffer contains binary data valid according to the pbefore
 996 parser specification, and ensures that, if successful, it contains binary data valid according to pafter
 997 on completion. Thus, Write is a simple parser-indexed parameterized state and error monad, that
 998 hides the mundane memory safety, binary layout, and error propagation details from its clients.
 999 Returning to example, we can define:

```
1000 (* A leaf writer for writing an integer *)
1001 val write_u32 : U32.t → Write unit parse_empty parse_u32
1002
1003 (* A generic higher-order framing operator, to be able to write two pieces of data in a row *)
1004 val frame (#a: Type) (#pframe #pafter : parser) (f: unit → Write a parse_empty pafter)
1005   : Write a pframe (parse_pair pframe pafter)
1006
1007 (* A lifting of the example_intro_mem lemma, with all binary layout details hidden, computationally a no-op. *)
1008 val write_example_correct : unit → Write unit (parse_pair parse_u32 parse_u32) parse_example
```

1009 This last lemma states that, if the output buffer contains valid data for parsing a pair of two integers,
 1010 then calling this function will turn that data into valid data for parsing an example struct value. With
 1011 these components, the user can now write their formatting code more succinctly, as shown below.
 1012 The output buffer, offsets, and error propagation are hidden in the effect and so, the user no longer
 1013 needs to explicitly reason about them. Furthermore, the code becomes much more self-explanatory.

```
1014 let write_example (x y : U32.t) : Write unit parse_empty parse_example =
1015   write_u32 x; frame (λ _ → write_u32 y); write_example_correct ()
```

1017 *Defining Write: A peek beneath the covers.* We represent Write using a dependent pair of indexed
 1018 monads ($p.\text{datatype}$ is the type of the values parsed by p):

```
1020 type repr (a : Type) pbefore pafter = (spec : repr_spec t pbefore.datatype pafter.datatype
1021   & repr_impl t pbefore pafter spec)
```

1022 The first field, spec, is a specificational parameterized monad evolving an abstract state from
 1023 pbefore.datatype to pafter.datatype. The second field is the Low* implementation, indexed by a
 1024 pair of parsers and the spec. As such, repr_impl is a parameterized-monad-indexed monad, or a
 1025 form of parameterized Dijkstra monad, a novel construction, as far as we are aware.

1027 *Compiling Write computations to C.* To compile a Write computation to C, or call it from other
 1028 Low* code, we simply reify it and project its Low* implementation:

1029

```

1030 let reify_spec (f: unit → Write a pbefore pafter)
1031   : repr_spec a pbefore.datatype pafter.datatype = fst (reify (f ()))
1032 (* Extract the Low* code of a computation to compile it to C *)
1033 let reify_impl (f: unit → Write a pbefore pafter)
1034   : repr_impl a pbefore pafter (reify_spec f) = snd (reify (f ()))
1035

```

Using effect subcomp for automatic parser rewriting. We can embed parser rewriting rules in the subcomp combinator for Write, as follows:

```

1038 val subcomp a (p1 : parser) (p2 p2' : parser{valid_rewrite p2 p2'}) (l_ : repr a p1 p2) : repr a p1 p2'
1039

```

where valid_rewrite p2 p2' is a relation on parser specifications stating that any binary data valid for p2 is also valid for p2'. Since the valid_rewrite goals can automatically be solved via SMT, this allows us to rewrite example as simply: (write_u32 x; frame (λ_ → write_u32 y)). The remaining overhead is framing; we eliminate it with another indexed effect layered on top of Write.

1045 5.2 Automated framing with FWrite

1046 Following the frame inference methodology proposed by Fromherz et al. [2021] in the context of a
 1047 concurrent separation logic, we define a new effect FWrite that automatically adds frames to the
 1048 computations when sequentially composing them.

```

1049 type frepr (a:Type) pbefore pafter = unit → Write a pbefore pafter
1050 val fbind (a b: Type) (p1 p1' p2 p2': parser)
1051   (frame_f: parser) (frame_g: parser)
1052   (l_ : squash (valid_rewrite (parse_pair frame_f p2) (parse_pair frame_g p1')))
1053   (f: frepr a p1 p2) (g: a → frepr b p1' p2')
1054   : frepr b (parse_pair frame_f p1) (parse_pair frame_g p2')
1055 effect { FWrite (t: Type) (pbefore pafter: parser) with { repr = frepr; bind = fbind; ... } }
1056

```

1057 The fbind combinator inserts frames frame_f and frame_g to the two computations f and g, and
 1058 adds a squashed goal to the VC ensuring that the framed postcondition of f, parse_pair frame_f p2,
 1059 can be rewritten into the framed precondition of g, parse_pair frame_g p1'. Under the hood, FWrite
 1060 computations are thunked Write computations; implementing fbind thus consists of composing
 1061 calls to f and g encapsulated by the frame combinator of the Write effect.

1062 To automatically infer frame_f and frame_g, and discharge the framing related VCs, similar
 1063 to Fromherz et al. [2021], we gather all the framing goals and implicits and discharge them using a
 1064 (partial) decision procedure that we implement as an F* tactic.

1065 With FWrite, we can now write example in, arguably, the most natural way:

```

1066 let write_example (x y : U32.t) : FWrite unit parse_empty parse_example = write_u32 x; write_u32 y
1067

```

1068 By successively layering several effects, we thus retrieve a proof style akin to proofs by refinement,
 1069 but for effectful computations. We abstract away reasoning about error handling, low-level byte
 1070 layout, and framing, through different indexed effects, finally providing a programmer with a
 1071 high-level interface closer to an idealized functional program to use verified low-level serializers.

1072 The FWrite effect scales to more than just writing a record of two integers. We show how to
 1073 write a variable-sized list of 32-bit integers, the list being prefixed by a header recording its size in
 1074 bytes. If p is a parser for the elements of the list, then parse_vllist p min max is a parser that first
 1075 reads a header consisting of an integer value that will be the total storage size of the list elements
 1076 in bytes, then checks that it is between min and max, then parses the list of elements using p for
 1077 each header. The min and max bounds are constants mandated by the data format and independent
 1078

1079 of the size of the actual output buffer. The following code writes a list of two integers, following
 1080 the data format specified by `parse_vllist`:

```
1081 let write_int_list (max_list_size: U32.t)
1082   : FWrite unit parse_empty (parse_vllist parse_u32 0 max_list_size) =
1083   write_vllist_nil parse_u32 max_list_size;
1084   write_u32 18ul; extend_vllist_snoc ();
1085   write_u32 42ul; extend_vllist_snoc ()
```

1087 The value of `max_list_size` constrains the size of the size header, but thanks to the abstraction
 1088 provided by `Write` (and hence `FWrite`), the user does not need to know about that actual size. The
 1089 code relies on two combinators:

```
1090
1091 val write_vllist_nil (p: parser) (max: U32.t) : FWrite unit parse_empty (parse_vllist p 0 max)
1092 val extend_vllist_snoc (#p: parser) (#min #max: U32.t) ()
1093   : FWrite unit (parse_pair (parse_vllist p min max) p) (parse_vllist p min max)
```

1094 `write_vllist_nil` starts writing an empty list by writing 0 as its size header. `extend_vllist_snoc`
 1095 assumes that the output buffer contains some variable-sized list immediately followed by an
 1096 additional element and “appends” the element into the list by just updating the size header of the
 1097 list; thus, the new element is not copied into the list, since it is already there at the right place.
 1098 `extend_vllist_snoc` also dynamically checks whether the size of the resulting list is still within the
 1099 bounds expected by the parser, returning an error if not.

1100 The data format specified by `parse_vllist p min max` and implemented by those `FWrite` combina-
 1101 tors corresponds to the format of variable-sized lists prefixed by their byte size as mandated by the
 1102 TLS 1.3 RFC [Rescorla 2018].

1104 5.3 Application: TLS 1.3 handshake extensions

1105 We have used the `Write` effect to generate the list of extensions of a TLS 1.3 [Rescorla 2018]
 1106 *ClientHello* handshake message, that a client sends to a server to specify which cipher suites and
 1107 other protocol extensions it supports. This is the most complex part of the handshake message
 1108 format, involving much more than just pairs: it involves variable-sized data and lists prefixed by
 1109 their size in bytes (as in the `write_int_list` example above), as well as tagged unions where the
 1110 parser of the payload depends on the value of the tag. Our `Write` effect based implementation of
 1111 *ClientHello* messages compiles to C and executes; we are currently rewriting it with `FWrite` to take
 1112 advantage of automated framing, and integrating it into a low-level rewriting of an implementation
 1113 of TLS in F^* [Bhargavan et al. 2013].

1114 A more powerful version of the `Write` and `FWrite` effects with support for Hoare-style pre- and
 1115 postconditions to prove functional correctness properties on the actual values written to the output
 1116 buffer, as well as error postconditions, in addition to correctness with respect to the data format, is
 1117 underway. With such an enhanced version, we plan to leverage pre- and postconditions to avoid
 1118 dynamic checks on writing variable-size list items.

1120 6 EXISTING APPLICATIONS OF INDEXED EFFECTS

1121 Indexed effects have been available in recent releases of F^* and have been used in Steel [Fromherz
 1122 et al. 2021] and DY^* [Bhargavan et al. 2021], two independent developments. These uses validate
 1123 our design and support our claim that indexed effects help structure effectful programs and proofs
 1124 at scale. We briefly summarize their work, while referring the reader to the Steel and DY^* papers
 1125 for more details.

1126
 1127

6.1 Steel

Steel is a language for developing and proving concurrent, dependently-typed F^* programs. Steel’s program logic is based on a shallow embedding of concurrent separation logic in F^* , while also enabling Hoare-style reasoning using a variant of heap predicates, called *selector predicates*.

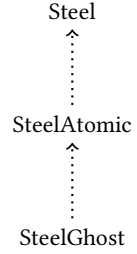
To enable smooth interoperation between separation logic and Hoare logic, Steel encodes pre- and postconditions as effect indices; we show a simplified version of the Steel effect representation:

```
let selpre fp = rheap fp → prop      let selpost fp a fp' = rheap fp → a → rheap fp' → prop
val steel_repr (a:Type) (fp:slprop) (fp':a → slprop) (req:selpre fp) (ens:selpost fp a fp') : Type
```

Separation logic specifications rely on assertions of type `slprop`, and are encoded through a precondition `fp`, and a return value dependent postcondition `fp'`. Similarly to the Hoare monad from §2.3, selector specifications consist of a precondition `req` and a 2-state postcondition `ens`. Note that `req` and `ens` operate on states (i.e., heaps) that are parameterized by the separation logic specifications: `rheap fp` is a restricted heap corresponding exactly to the predicate `fp`. This is another instance of a hybrid, dependent indexing structure, where the `fp` indices constrain the selector predicates.

To reason about concurrent programs, Steel also models atomic computations, as well as a notion of ghost state, which can be manipulated through `ghost`, computationally irrelevant computations. Ghost and atomic computations are separated from generic Steel functions, and are thus modeled as their own effects, `SteelGhost` and `SteelAtomic`, with two additional effect indices to encode verification conditions related to atomicity. Nevertheless, a ghost computation can always be seen as atomic, while an atomic computation is but a special case of a generic Steel computation. Steel captures this hierarchy through lifts between its different effects, which are automatically inserted by our framework when needed.

In Steel, indexed effects thus provide a foundation to structure reasoning, enabling, for instance, a separation of verification conditions related to atomicity and separation logic. Leveraging this structure, Steel automates framing reasoning, using a methodology similar in spirit to the one presented in §5.2, albeit applied to a full-fledged, impredicative, concurrent separation logic, which simplified the development of a wide variety of verified libraries, ranging from self-balancing trees and concurrent queues to 2-party session types.



6.2 DY^*

DY^* is an F^* -based framework for symbolic verification of security protocols and has been used for the first symbolic analysis of the Signal protocol, the messaging protocol used in WhatsApp, while accounting for an unbounded number of ratcheting rounds.

Protocols sessions in DY^* are modeled as partial, stateful F^* functions that may raise exceptions and the underlying state is a global, monotonic trace that tracks the interleaved execution of sessions. This is represented as an indexed effect for a state and exception Dijkstra monad, called `Crypto`. All the security protocols in DY^* , including Signal, are written in the `Crypto` effect, and verified against the trace-based properties expressed as specifications in the Dijkstra monad.

```
type wp (a:Type) = (option a → trace → prop) → trace → prop
```

(* The monotonicity property of the trace is internalized in the repr via extends *)

```
type crypto_repr (a:Type) (w:wp a) =
```

```
  s0:trace → PURE (option a & trace) (λ p → w (λ x s1 → s1 `extends` s0 ⇒ p (x, s1)) s0)
```

Without indexed effects, defining such an effect in F^* would not be possible, and using an axiomatized effect for verifying security properties is clearly suboptimal. Though one could derive an effect for state and exceptions using the Dijkstra Monads for Free methodology [Ahman et al. 2017], it does not allow internalizing trace monotonicity, as they have done here.

7 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. Filinski [1999] presents a framework for specifying and implementing layered monads, focusing on implementing them uniformly with delimited continuations and state, though does not consider indexed monads. Tate’s (2013) productors 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, though neither handle dependently typed programs. Bracker and Nilsson [2015] provide a Haskell plugin for polymonad programming—our implementation also provides support for polymonads, where not only the indices but also the effect label can vary among the computation arguments to bind, a feature used in Steel [Fromherz et al. 2021]. 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.

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 1)—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. By packaging our trees into an effect, we benefit from automatic 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 we 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 proofs of stateful programs. Besides, with 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 benefit from them when programming with effects in a dependently typed language, we have designed

1226 and implemented *indexed effects* as a feature of F^* . In doing so, we have simplified F^* 's core logic,
 1227 while also enabling new abstractions for programming and proving. Being available in F^* for the
 1228 past year, we have already seen indexed effects deployed by users in various settings, giving us
 1229 confidence that our work scales to large developments. By lowering the bar to programming with
 1230 indexed monads, we hope to encourage the development of new indexed constructions and new
 1231 patterns of proof for effectful software.

1232

1233

1234 REFERENCES

- 1235 Danel Ahman, Cédric Fournet, Cătălin Hrițcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness:
 1236 Foundations and Applications of Monotonic State. *PACMPL* 2, POPL (jan 2018), 65:1–65:30. <https://arxiv.org/abs/1707.02466>
- 1237 Danel Ahman, Cătălin Hrițcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and
 1238 Nikhil Swamy. 2017. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages*
 1239 (*POPL*). ACM, 515–529. <https://doi.org/10.1145/3009837.3009878>
- 1240 Robert Atkey. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19 (2009), 335–376. Issue 3-4.
 1241 <https://doi.org/10.1017/S095679680900728X>
- 1242 Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*
 1243 84, 1 (2015), 108–123. <https://doi.org/10.1016/j.jlamp.2014.02.001>
- 1244 Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations. In *Program-*
 1245 *ming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings (Lecture*
 1246 *Notes in Computer Science, Vol. 4279)*, Naoki Kobayashi (Ed.). Springer, 114–130. https://doi.org/10.1007/11924661_7
- 1247 Karthikeyan Bhargavan, Abhishhek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim
 1248 Würtele. 2021. DY*: Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. (2021).
 1249 <https://publ.sec.uni-stuttgart.de/bhargavanbichavatdohosseynikuestersschmitzwuertele-eurosp-2021.pdf> To appear.
- 1250 Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Cătălin Hrițcu, Samin
 1251 Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko,
 1252 Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-
 1253 Béguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd*
 1254 *Summit on Advances in Programming Languages*. <http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-2017-1.pdf>
- 1255 Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and P. Strub. 2013. Implementing TLS with
 1256 verified cryptographic security. In *IEEE Symposium on Security and Privacy*. 445–459.
- 1257 Jan Bracker and Henrik Nilsson. 2015. Polymonad Programming in Haskell. In *Proceedings of the 27th Symposium on the*
 1258 *Implementation and Application of Functional Programming Languages* (Koblenz, Germany) (*IFL '15*). Association for
 1259 Computing Machinery, New York, NY, USA, Article 3, 12 pages. <https://doi.org/10.1145/2897336.2897340>
- 1260 Edwin C. Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN International*
 1261 *Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo
 1262 Uustalu (Eds.). ACM, 133–144. <https://doi.org/10.1145/2500365.2500581>
- 1263 Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on*
 1264 *Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science, Vol. 4963)*.
 1265 Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
- 1266 Andrzej Filinski. 1999. Representing Layered Monads. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Program-*
 1267 *ming Languages*. ACM, 175–188. <https://doi.org/10.1145/292540.292557>
- 1268 Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananan-
 1269 dro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. In *Proceedings of*
 1270 *the International Conference on Functional Programming (ICFP)*.
- 1271 Michael Hicks, Gavin M. Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. 2014. Polymonadic Programming. In
 1272 *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12*
 1273 *April 2014 (EPTCS, Vol. 153)*, Paul Levy and Neel Krishnaswami (Eds.). 79–99. <https://doi.org/10.4204/EPTCS.153.7>
- 1274 Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In
 1275 *Proceedings of the 14th International Conference on Formal Methods (Hamilton, Canada) (FM'06)*. Springer-Verlag, Berlin,
 1276 Heidelberg, 268–283. https://doi.org/10.1007/11813040_19
- 1277 Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *The 41st Annual ACM SIGPLAN-*
 1278 *SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh
 1279 Jagannathan and Peter Sewell (Eds.). ACM, 633–646. <https://doi.org/10.1145/2535838.2535846>

1274

- 1275 Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (Aug. 2015), 94–105.
 1276 <https://doi.org/10.1145/2887747.2804319>
- 1277 Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN*
 1278 *Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna
 1279 and Andrew D. Gordon (Eds.). ACM, 486–499. <http://dl.acm.org/citation.cfm?id=3009872>
- 1280 Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2.
 1281 Springer.
- 1282 Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN*
 1283 *Symposium on Principles of Programming Languages (Paris, France) (POPL 2017)*. Association for Computing Machinery,
 1284 New York, NY, USA, 500–514. <https://doi.org/10.1145/3009837.3009897>
- 1285 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hrițcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra
 1286 Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (July 2019), 29 pages. <https://doi.org/10.1145/3341708>
- 1287 Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hrițcu, Monal
 1288 Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem
 1289 Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European*
 1290 *Symposium on Programming (ESOP)*. Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2
- 1291 Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium*
 1292 *on Logic in Computer Science (LICS '89)*, Pacific Grove, California, USA, June 5-8, 1989. IEEE Computer Society, 14–23.
 1293 <https://doi.org/10.1109/LICS.1989.39155>
- 1294 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J.*
 1295 *Funct. Program.* 18, 5-6 (2008), 865–911. <http://ynot.cs.harvard.edu/papers/jfsep07.pdf>
- 1296 Dominic Orchard, Philip Wadler, and Harley Eades III. 2020. Unifying graded and parameterised monads. In *Proceedings*
 1297 *Eighth Workshop on Mathematically Structured Functional Programming, MSFP at ETAPS 2020, Dublin, Ireland, 25th April*
 1298 *2020 (EPTCS, Vol. 317)*, Max S. New and Sam Lindley (Eds.). 18–38. <https://doi.org/10.4204/EPTCS.317.2>
- 1299 Dominic A. Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM*
 1300 *SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 13–24. <https://doi.org/10.1145/2633357.2633368>
- 1301 Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1
 1302 (2003), 69–94. <https://doi.org/10.1023/A:1023064908962>
- 1303 Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th*
 1304 *European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice*
 1305 *of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe
 1306 Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7
- 1307 Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-
 1308 Béguelin, Antoine Delignat-Lavaud, Cătălin Hrițcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017.
 1309 Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. <https://doi.org/10.1145/3110261>
- 1310 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan
 1311 Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the*
 1312 *28th USENIX Conference on Security Symposium (Santa Clara, CA, USA) (USENIX Security 2019)*. USENIX Association,
 1313 USA, 1465–1482.
- 1314 E. Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446. <https://tools.ietf.org/html/rfc8446>
- 1315 Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. 2011. Lightweight monadic programming in ML. In *Proceeding*
 1316 *of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21,*
 1317 *2011 (ICFP '11)*. 15–27. <https://www.cs.umd.edu/~mwh/papers/swamy11monad.html>
- 1318 Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bharga-
 1319 van, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin.
 1320 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of*
 1321 *Programming Languages (POPL)*. ACM, 256–270. <https://www.fstar-lang.org/papers/mumon/>
- 1322 Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore:
 1323 An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.* 4,
 1324 ICFP, Article 121 (Aug. 2020), 30 pages. <https://doi.org/10.1145/3409003>
- 1325 Nikhil Swamy, Joel Weinberger, Cole Schlessinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs
 1326 with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design*
 1327 *and Implementation (PLDI '13)*. 387–398. <https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/>

- 1324 Ross Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the 40th Annual ACM SIGPLAN-*
1325 *SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) (POPL '13). Association for Computing
1326 Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/2429069.2429074>
- 1327 Philip Wadler. 1992. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-*
1328 *SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi
1329 Sethi (Ed.). ACM Press, 1–14. <https://doi.org/10.1145/143165.143169>
- 1330
- 1331
- 1332
- 1333
- 1334
- 1335
- 1336
- 1337
- 1338
- 1339
- 1340
- 1341
- 1342
- 1343
- 1344
- 1345
- 1346
- 1347
- 1348
- 1349
- 1350
- 1351
- 1352
- 1353
- 1354
- 1355
- 1356
- 1357
- 1358
- 1359
- 1360
- 1361
- 1362
- 1363
- 1364
- 1365
- 1366
- 1367
- 1368
- 1369
- 1370
- 1371
- 1372