



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

1 Programming and Proving with Indexed Effects

2
3 ASEEM RASTOGI, Microsoft Research, India
4
5 GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina
6
7 AYMERIC FROMHERZ, Microsoft Research & CMU, USA
8 TAHINA RAMANANANDRO, Microsoft Research, USA
9 NIKHIL SWAMY, Microsoft Research, USA

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

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

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

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

22 1 INTRODUCTION

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

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

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

47
48 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 $\{I, \oplus, e\}$, whose `return` and `bind` have the following signatures: $\text{return} : a \rightarrow G a e$ 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 memory
 54 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 *Hoare monads.* Working in a dependently typed setting, and aiming initially to prove stateful
 62 programs correct in Coq, Nanevski et al. [2008] developed the Hoare monad, a typeclass H , in-
 63 dexed by memory predicates $p : \text{mem} \rightarrow \text{prop}$ and $q : a \rightarrow \text{mem} \rightarrow \text{prop}$, with combinators of the form
 64 $\text{return} : x : a \rightarrow H a (p x) p$, meaning that a pure computation returns an $x : a$ while preserving any pred-
 65 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
 66 structure represents the rule for sequential composition in Hoare logics. Encoding a Hoare logic in
 67 the indices is a powerful concept, and Hoare monads have been used to prove the correctness of
 68 many programs in a variety of program logics. But, even among Hoare monads, several variants
 69 exist. For example, in some versions, the postcondition q is a relation on a pair of memory states,
 70 i.e., $q : \text{mem} \rightarrow a \rightarrow \text{mem} \rightarrow \text{prop}$.

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

83 1.1 New hybrid constructions

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

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

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

101 **1.2 Type-and-effect directed elaboration**

103 Lacking a typeclass for our structures, we develop a new language feature to support a type-and-
 104 effect directed elaboration of source programs written in a direct, applicative syntax into any indexed
 105 monad structure. Our feature, *indexed effects*, is usable with any monad-like type constructor L ,
 106 with an arbitrary number of indices, and combinators $\text{return} : x:a \rightarrow L a \vec{i}$ and $\text{bind} : L a \vec{i} \rightarrow (x:a \rightarrow$
 107 $L b \vec{j}) \rightarrow L b \vec{k}$ —note the indices may vary arbitrarily in return and bind . Given such a signature, our
 108 algorithm allows programs to be developed in an ML-like applicative syntax, while elaborating them
 109 automatically into the underlying monad-like combinators on L . We have implemented indexed
 110 effects in F^* , enhancing its user-defined effect system, previously limited to Dijkstra monads only,
 111 to cover all forms of indexed monads.

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

```
115 let state = int      type tag = R | RW      let (⊕) t0 t1 = match (t0, t1) with | (R, R) → R | _ → RW
116 let gst (a:Type) (t:tag) = s0:state → r:(a & state) { t=R ==> s0 ==> snd r }
117 let return (x:a) : gst a R = λs → x,s
118 let bind (f:gst a t0) (g: a → gst b t1) : gst b (t0 ⊕ t1) = λs0 → let x, s1 = f s0 in g x s1
119 let read () : gst state R = λs → s,s    let write (s:state) : gst unit RW = λ_ → (), s
```

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

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

```
126 let subcomp (f:gst a t0 { ∃t. t1 == t0 ⊕ t }) : gst a t1 = f
127 let if_then_else (f:gst a t0) (g:gst a t1) (_:bool) = gst a (t0 ⊕ t1)
128 effect { GST (a:Type) (t:tag) with { repr = gst; return; bind; read; write; subcomp; if_then_else } }
```

129 130 With these definitions in place, one can write $\text{if } b \text{ then } (\text{write} (\text{read}() + 1); 0) \text{ else } \text{read}()$, while
 131 the framework infers the computation type GST int RW and internally elaborates the program into
 132 the following explicitly monadic form:

```
133 if b then subcomp (bind (read()) (λ x → bind (write (x + 1)) (λ _ → return 0))) else subcomp (read())
```

134 135 Effect definitions can also be layered, e.g., we could add a layer to represent exceptions on top of
 136 the GST effect, with implicit coercions to move between them.

137 **1.3 Formalization of indexed effects and simplification to the theory of F^***

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

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

145 146 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 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.

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

209 *Basic syntax.* F^* syntax is roughly modeled on OCaml (`val`, `let`, `match`, etc.). Binding occurrences
 210 b take the form $x:t$, declaring a variable x at type t ; or $\#x:t$ indicating that the binding is for an
 211 implicit argument. The syntax $\lambda b_1 \dots b_n \rightarrow t$ introduces a lambda abstraction (metavariable t ranges
 212 over both types and terms), whereas $b_1 \rightarrow \dots \rightarrow b_n \rightarrow c$ is the shape of a curried function type with
 213 *computation type* c (more about them shortly). Refinement types are written $b\{t\}$, e.g., the type
 214 $x:\text{int}\{x \geq 0\}$ represents natural numbers. We define `squash` t as the unit refinement $_:\text{unit}\{t\}$, which
 215 can be seen as the type of (computationally irrelevant) proofs of t . As usual, we omit the type in a
 216 binding when it can be inferred; and for non-dependent function types, we omit the variable name.
 217 E.g., the type $\#a:\text{Type} \rightarrow \#m:\text{nat} \rightarrow \#n:\text{nat} \rightarrow \text{vec } a \ m \rightarrow \text{vec } a \ n \rightarrow \text{vec } a \ (m + n)$ represents the append
 218 function on vectors, where the two explicit arguments and the return type depend on the three
 219 implicit arguments marked with ‘#’. We mostly omit implicit binders, except when needed for
 220 clarity, treating all unbound variables in types as prenex quantified, writing the type of append as
 221 just $\text{vec } a \ m \rightarrow \text{vec } a \ n \rightarrow \text{vec } a \ (m + n)$. We also omit universe annotations.

222 Returning to the computation types c , F^* distinguishes computations from values in a manner
 223 similar, though not identical, to Levy’s (2004) Call-By-Push-Value calculus. Computation types
 224 include `Tot` t ($x:t_1 \rightarrow t_2$ is a shorthand for $x:t_1 \rightarrow \text{Tot } t_2$) for pure, total computations. Another built-
 225 in computation type is `Lemma` (`requires` p) (`ensures` q), which is the type of a computation which
 226 when executed in a context validating p terminates in a context validating q , i.e., it can be seen as
 227 sugar for `squash` $p \rightarrow \text{squash } q$. When p is trivial, we simply write `Lemma` (`ensures` q) or `Lemma` q .

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

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

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

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

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

249 `effect { ST (a:Type) (s:Type) with { repr = st; return; bind; get; put } }`
 250

251 We use the term “effect” to refer to a computation type constructor. Here, ST is an effect and the
 252 definition above introduces a new user-defined computation type, ST a s, whose underlying representation
 253 is st a s, supporting the return and bind combinators, and two actions ST?.get : unit → ST s s
 254 and ST?.put : s → ST unit s—the notation ST?.op names the operation op in the ST effect declaration.⁴
 255

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

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

266 2.2 Indexed effects, in a nutshell

267 An effect declaration in F[★] allows promoting any indexed monad m into an effect M. Doing so
 268 requires:

- 269 (1) A representation type, m t i, for an arbitrary arity |i|.
- 270 (2) A return, whose type is of the form x:a → m a p̄, for some p̄.
- 271 (3) A bind, whose type is of the form m a p̄ → (x:a → m b q̄) → m b r̄, for some p̄, q̄, r̄
- 272 (4) Zero or more actions, ā where each a_i has a type of the form x_i:t_i → m s_i p̄_i.
- 273 (5) An optional subsumption combinator, subcomp, _:m a p̄ { pre } → m a q̄, which allows re-indexing
 274 an m a p̄ to an m a q̄, when pre : prop is valid.
- 275 (6) An optional branching combinator, if_then_else whose type has the form m a p̄ → m a q̄ →
 276 bool → Type, such that for all x:m a p̄, y:m a q̄, and b:bool, if b then subcomp x else subcomp y
 277 has type if_then_else x y b.

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

285 *Composing multiple effects.* Further, given two effect declarations M and N with representation
 286 types m and n, F[★]’s effect system supports implicitly lifting M-computations to N-computations if the
 287 programmer supplies a combinator lift: _:m a i { pre } → n a j.

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

291
 292 ⁴Strictly speaking, to reference the actions in the GST effect of §1.2, we should have written GST?.read and GST?.write,
 293 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 **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) }`
 303

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 _ → T) (λ s0 r s1 → s0 == s1 ∧ r == x) = λ s → 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 ∧ (forall x s1. q_f s0 x s1 == p_g x s1)) (λ s0 r s2 → (exists x s1. q_f s0 x s1 ∧ q_g x s1 r s2))`
 310 `= λ s0 → 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 _ → T) (λ s0 x s1 → s0 == s1 ∧ x == s1) = λ s → s, s`
 326 `let put (x:s) : hst unit s (λ _ → T) (λ _ s1 → x == s1) = λ _ → (), x`

328 *Subsumption, or the Hoare rule of consequence.* Hoare logics typically include a rule of consequence,
 329 enabling preconditions to be strengthened and postconditions to be weakened. Our Hoare logic
 330 encoded in hst also admits such a rule, which we can encode as a subsumption combinator for
 331 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' = (forall s. p' s == p s) ∧ (forall 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)`
 341

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.
 353

```
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

```
362  let mod_or_sqrt (b:bool) (x y:loc) : HST unit store
363      (requires λs → sel s x > 0)
364      (ensures λs0 _ s1 → ∃v. (if b then v ≤ sel s0 x else v ≥ sel s0 x) ∧ s1 == upd s0 y v)
365      = if b then write y (read y % read x) else write y (read x * read x)
```

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

374 Without the effect-based elaboration, one could try to write a program directly in the `hst` monad.
 375 It would look something like this (where `read_hst` and `write_hst` are the `hst` analogs of `read` and
 376 `write`), where one essentially has to build a Hoare-style derivation by hand. Even if the system is
 377 able to infer all the missing implicit arguments (shown as underscores), which it cannot, in this
 378 case, this style is verbose and obscures the program beyond recognition.
 379

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

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 $p : a \rightarrow prop$, return a $v : a$ validating
 399 $p\ v$. In this case, we have a term $v0 \% v1$ which, due to a possible division by zero, has type
 400 `PURE` `int` $(\lambda p \rightarrow v1 \neq 0 \wedge (v1 \neq 0 \implies p(v0 \% v1)))$, indicating that it is only safe run the term in context
 401 where $v1 \neq 0$ 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
```

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

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

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

```
433 let label = set loc      let flow = label & label      let flows = list flow
434 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 }
```

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

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

```
446 val return (x:a) : hifc a {} {} [] ( $\lambda \_ \rightarrow \top$ ) ( $\lambda s0 \ r \ s1 \rightarrow s0 == s1 \wedge r == x$ )
447
448 val bind (f:hifc a r0 w0 fs0 p q) (g: (x:a  $\rightarrow$  hifc b r1 w1 fs1 (r x) (s x)))
449   : hifc b (r0  $\cup$  r1) (w0  $\cup$  w1) (fs0 @ add_source r0 (({}, w1)::fs1))
450   ( $\lambda s0 \rightarrow p \ s0 \wedge (\forall x \ s1. \ q \ s0 \ x \ s1 \wedge \text{modifies } w0 \ s0 \ s1 \implies r \ x \ s1)$ )
451   ( $\lambda s0 \ r \ s2 \rightarrow (\exists x \ s1. (q \ s0 \ x \ s1 \wedge \text{modifies } w0 \ s0 \ s1) \wedge (s \ x \ s1 \ r \ s2 \wedge \text{modifies } w1 \ s1 \ s2))$ )
452 where add_source r fs = List.map ( $\lambda (src, sink) \rightarrow (src \cup r, sink)$ ) fs and @ is list concatenation
```

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

462 We can show that the triple of additional indices of HIFC, (`reads`, `writes`, `flows`) form a monoid
 463 (under a suitable equivalence relation) whose unit is `{}.`, `{}.`, `[]` and whose elements can be composed
 464 with \oplus (shown below), which is the indexing pattern of a graded monad, used on `return` and `bind`.
 465

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

467 Packaging `hifc` as an effect HIFC, together with a subsumption relation that allows widening the
 468 reads, writes and flows sets together with the Hoare rule of consequence; a branching combinator;
 469 and actions to read and write individual locations, allows us to write and prove effectful programs for
 470 both correctness and security, by reasoning only about their indices. For example, `write l1(read l0)`
 471 is inferred to have type HIFC unit $\{l_0\} \{l_1\} [\{l_0\}, \{l_1\}] (\lambda _ \rightarrow \top) (\lambda s0 _ s1 \rightarrow \text{sel } s_1 l_1 == \text{sel } s_0 l_0)$.
 472

473 *Refining flows with Hoare reasoning.* Monadic label-based information flow is inherently imprecise,
 474 since it conflates data and control dependence. To illustrate, consider `read h; write l` (`read l + 1`)
 475 which has the type HIFC unit $\{h, l\} \{l\} [\{h\}, \{l\}] (\lambda _ \rightarrow \top) (\lambda s0 _ s1 \rightarrow \text{sel } s_1 l = \text{sel } s_0 l + 1)$, suggesting
 476 that it leaks information from `h` to `l`, when in reality no such flow exists since the `read h` is redundant.
 477 However, HIFC's pre- and postconditions can be used to recover precision. The re-indexing coercion,
 478 `refine`, allows removing a flow `f` from HIFC $a \ r \ w (f::fs) \ p \ q$ when the Hoare specifications `p` and `q`
 479 allow proving that the `f`-flow is spurious.

```
480 val refine (_:(unit  $\rightarrow$  HIFC a r w (f::fs) p q) {(\forall from to v. from  $\in$  fst f  $\wedge$  to  $\in$  snd f  $\wedge$  from  $\neq$  to  $\implies$ 
481   ( $\forall s0 \ x \ x' s1 s1'. (p \ s0 \wedge p \ (\text{upd } s0 \ \text{from } v) \wedge q \ s0 \ x \ s1 \wedge q \ (\text{upd } s0 \ \text{from } v) \ x' s1' \wedge$ 
482    $\text{modifies } w \ s0 \ s1 \wedge \text{modifies } w \ (\text{upd } s0 \ \text{from } v) \ s1') \implies$ 
483    $\text{sel } s1 \ \text{to} == \text{sel } s1' \ \text{to})})} : \text{unit} \rightarrow \text{HIFC } a \ r \ w \ fs \ p \ q$ 
```

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

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

```

491     constant    $T ::= \text{unit} \mid \text{Type}_u \mid \text{sum} \mid \text{inl} \mid \text{inr}$     computation type    $C ::= \text{Tot } t$ 
492     term       $e, t ::= x \mid T \mid x:t_1\{t_2\} \mid x:t \rightarrow C \mid \lambda x:t. e \mid e_1 e_2 \mid \text{case}_C(e \text{ as } y) x.e_1 x.e_2$ 
493
494
495
496

```

Fig. 1. Syntax of TotalF *

497 structures. Our point is that by designing hybrid indexed monads, one can obtain new reasoning
498 principles whose combination is more powerful than their components.
499

500 3 FORMALIZATION AND IMPLEMENTATION OF INDEXED EFFECTS

501 In this section we provide a semantics for indexed effects via a type-and-effect directed, type-
502 preserving translation to a core lambda calculus with dependent and refinement types. The essence
503 of the translation is that it unfolds a computation type into its underlying indexed monadic
504 representation. As a result, while the programmers may use indexed effects to build abstractions
505 for the ease of programming and proving, the structure of types still provides guidance as to how
506 the programs execute.

507 This translation semantics of indexed effects also allows us to minimize the formal core of F * .
508 Specifically, we can remove all the effectful features from it, including currently primitive Dijkstra
509 monads, as they can simply be encoded. This in itself is a significant advancement, since any further
510 metatheory of F * 's core no longer has to consider Dijkstra monads. The translation semantics may
511 also provide a way for other proof assistants, many of which have core calculi resembling TotalF * ,
512 to implement indexed effects, though some of the details, like refinement types, may differ.

513 We begin by describing the target language of the translation, and the new core calculus of
514 F * , that we call TotalF * . We then present Indexed Monadic F * (IMF *), a surface language with
515 user-defined indexed effects, and present a syntax-directed, type-and-effect driven translation
516 from IMF * to TotalF * ; we prove that the translation is type-preserving implying that the typing
517 derivations in IMF * can be soundly interpreted in TotalF * . We also discuss several implementation
518 aspects of indexed effects in the F * typechecker.
519

520 3.1 TotalF * : The new core calculus for F *

521 Figure 1 shows the syntax of TotalF * . It includes proof-irrelevant refinement types, dependent
522 functions, dependent pattern matching with case (with explicitly annotated return computation type
523 C), and a non-cumulative predicative universe hierarchy (Type $_u$). We elide universe annotations in
524 the rest of the section. TotalF * has one computation type Tot t for total computations. The calculus
525 is essentially an effectless fragment of EMF * [Ahman et al. 2017], the previous F * core. Though
526 terms and types are in the same syntactic class in TotalF * , we use e and t to distinguish them, when
527 it promotes clarity. Similarly, we use x as term variables, and a and b as type variables.
528

529 The most distinctive feature of TotalF * is its use of a subtyping relation in type conversion.
530 Definitionally equal types are convertible. Subtyping also includes types related by refinement,
531 where $\Gamma \models t$ is a proof-irrelevant logical entailment, implemented in F * using an SMT solver, or
532 handled by user-provided tactics.
533

$$\frac{t_0 \rightarrow^* t \quad t_1 \rightarrow^* t \quad \Gamma \vdash t_0 <: t'_0 \quad \Gamma, x:t_0\{t_1\} \models t'_1}{\Gamma \vdash t_0 <: t_1 \quad \Gamma \vdash x:t_0\{t_1\} <: x:t'_0\{t'_1\}}$$

536 TotalF * is still simpler than the logical core of the full F * system, lacking inductive type definitions
537 and general pattern matching, recursive functions, and universe polymorphism. Importantly,
538 TotalF * lacks F * 's equality reflection, which is crucial to F * 's extensional type theory. We plan to
539

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

Fig. 2. Syntax of IMF *

551 study these enhancements in the future, grateful to no longer have to consider their interaction
552 with Dijkstra monads.
553

554 3.2 IMF * : A surface language with user-defined indexed effects

555 IMF * models a surface language with support for user-defined indexed effects. Figure 2 shows the
556 syntax. IMF * adds effectful constructs to TotalF * . These include computation types $F t \bar{e}$ (where F
557 is the effect, t is the return type, and \bar{e} are the effect indices) in C , let bindings, and reify and reflect
558 coercions between the computation types and their underlying representations.
559

560 An effect definition D defines an indexed effect F with indices types $\overline{x : t}$ and combinators
561 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
562 (effect M ranges over Tot and F). Our implementation allows for specifying an optional custom
563 effect combinator for combining the branches of case —we discuss it in §3.5.

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

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

569 To illustrate the typing rules, we use the graded state monad from Section 1, partly reproduced
570 here, as a running example. One difference, to illustrate the use of reflect, is that rather than
571 including the read and write actions as part of the effect definition, we show a desugared form
572 where we use $\text{GST}.\text{reflect}$ to promote a gst a t to a GST a t computation type, separately from the
573 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)
```

580
581 *Typechecking effect definitions and lifts.* While IMF * does not impose any constraints on the
582 layering or indexing structure of the effects, the types of the combinators in an effect definition
583 D are constrained to have specific shapes, as described in §1.2. Whereas previously we left the
584 additional index arguments in these combinators implicit, here, to be clearer, we make them explicit.
585 We write $F.\text{repr}$, $F.\text{bind}$ etc. to denote e_{repr} , e_{bind} etc. for the effect F in an ambient signature S .

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

| | | | | |
|-----|--|---|---|--|
| 589 | | T-LET | | |
| 590 | T-VAR | $\Delta \vdash e_1 : C_1 \rightsquigarrow e'_1$ | $\Delta \vdash \lambda x. e_2 : x:C_1^t \rightarrow C_2 \rightsquigarrow e'_2$ | $x \notin FV(C_2^t)$ |
| 591 | $x : t \in \Gamma$ | $\Delta, f : \hat{C}_1, g : x:C_1^t \rightarrow \hat{C}_2 \vdash C.bind (lift_{C_1^t}^C f) (\lambda x. lift_{C_2^t}^C (g x)) : \hat{C} \rightsquigarrow e'$ | | |
| 592 | $\Delta \vdash x : t \rightsquigarrow x$ | | $\Delta \vdash \text{let } x = e_1 \text{ in } e_2 : C \rightsquigarrow e'[e'_1/f][e'_2/g]$ | |
| 593 | | | | |
| 594 | T-REFIFY | $\Delta \vdash e : C \rightsquigarrow e'$ | T-REFLECT | T-SUB |
| 595 | | | $\Delta \vdash e : \hat{C} \rightsquigarrow e'$ | $\Delta \vdash e : C' \rightsquigarrow e' \quad \Delta \vdash C' <: C$ |
| 596 | | $\Delta \vdash \text{reify } e : \hat{C} \rightsquigarrow e'$ | $\Delta \vdash F.\text{reflect } e : C \rightsquigarrow e'$ | $\Delta \vdash e : C \rightsquigarrow e'$ |
| 597 | | | | |
| 598 | T-CASE | | | |
| 599 | $\Delta \vdash e : \text{sum } t_1 t_2 \rightsquigarrow e' \quad \Delta, y : \text{sum } t_1 t_2 \vdash C : \text{Type} \rightsquigarrow t' \quad i \in 1, 2 \quad T = \text{inl if } i = 1, \text{inr o/w}$ | | | |
| 600 | $\Delta, x : t_i \vdash e_i : C_i[T x/y] \rightsquigarrow e'_i \quad \Delta, x : t_i, f_i : \hat{C}_i[T x/y] \vdash lift_{C_i^t}^C f_i : \hat{C}_i[T x/y] \rightsquigarrow e''_i$ | | | |
| 601 | | | | |
| 602 | $\Delta \vdash \text{case}_C(e \text{ as } y) x.e_1 x.e_2 : C[e/y] \rightsquigarrow \text{case}_{C'}(e' \text{ as } y) x.e''_1[e'_1/f_1] x.e''_2[e'_2/f_2]$ | | | |
| 603 | | | | |
| 604 | C-M | SC-M | S-SUB | |
| 605 | $\Delta \vdash \hat{C} : \text{Type} \rightsquigarrow t'$ | $\Delta \vdash C_1 : \text{Type} \rightsquigarrow _ \quad \Delta \vdash t_1 : \text{Type} \rightsquigarrow t'_1 \quad \Delta \vdash t_2 : \text{Type} \rightsquigarrow t'_2$ | $\Delta \vdash \hat{C} <: \hat{C}_1 \quad \Delta \vdash t'_1 <: t'_2$ | |
| 606 | | $\Delta \vdash \hat{C} <: \hat{C}_1$ | | |
| 607 | $\Delta \vdash C : \text{Type} \rightsquigarrow \text{Tot } t'$ | $\Delta \vdash C <: C_1$ | | $\Delta \vdash t_1 <: t_2$ |
| 608 | | | | |

Fig. 3. IMF[★] typing judgments (primed symbols are TotalF[★] syntax)

611 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
612 $F.\text{subcomp}$ combinator and the related if_then_else combinator later.

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

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

623 Finally, an expression e defining a lift from F to F' is typechecked as a coercion from $F.\text{repr}$ to
624 $F'.\text{repr}$, 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
625 IMF[★] gets an automatic lift from Tot : $\text{lift}_{\text{Tot}}^F = F.\text{return}$. We discuss checks we impose on the lifts
626 collectively to ensure coherence in §3.5.

627 3.3 Type-and-effect directed elaboration

629 The typing rules from Figure 3 elaborate the implicitly monadic IMF[★] terms to TotalF[★] by inserting
630 binds and lifts. The placement of these combinators are purely syntax-directed, since computation
631 types can only appear immediately to the right of an arrow, and because we enforce left-to-right
632 evaluation order. However, applying the bind and lift combinators requires inference of the effect
633 indices and the combinator arguments (e.g., for the $\overline{x : t}$ binders in the combinator types). For this,
634 the system includes a declarative, non-coercive subtyping rule, and implicit arguments in all the
635 rules are chosen nondeterministically. In §3.5, we discuss how this nondeterminism is resolved
636 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 $C = \text{Tot } _, \text{bind}$ 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 $f : \hat{C}_1$ and $g : x:C_1^t \rightarrow \hat{C}_2$, applies the lift combinator 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 `let x = read () in write x+1`, the rule T-LET first typechecks `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 combinator 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 \models 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.

687 property for its refinement logic—these results also apply to its TotalF[★] fragment. The proof of the
 688 theorem is by mutual induction on the typing derivation with the following lemmas:

689 LEMMA 3.2 (COMMUTATION OF SUBTYPING).

690 (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 $[\Gamma]_S \vdash t' <: t'_1$.*
 691 (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 $[\Gamma]_S \vdash t' <: t'_1$.*

692 The essence of effect abstraction: Admissibility of using $F.\text{subcomp}$ for subtyping. The reader
 693 may have noticed that the rule SC-M breaks the effect abstraction by peeking into the effect
 694 representation for checking subtyping. However, this is not necessary: we show the admissibility
 695 of checking subtyping using the $F.\text{subcomp}$ effect combinator.
 696

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

699 $S; \cdot \vdash F.\text{subcomp} : a : \text{Type} \rightarrow \overline{x : t} \rightarrow f : F.\text{repr } a \bar{e}\{t_1\} \rightarrow F.\text{repr } a \bar{e}_1$
 700 and

701 $F.\text{subcomp} = \lambda a \bar{x} f. f$

702 where t_1 is a refinement formula. The intuition is that the combinator is a coercion that can be
 703 used to coerce a computation from $F t \bar{e}$ to $F t \bar{e}_1$, provided that the refinement formula is valid.
 704 The implementation of $F.\text{subcomp}$ is required to be an identity function. This is because subtyping
 705 in F[★] is intentionally non-coercive, so that the application of subtyping does not disturb equality.
 706 To prove $F t \bar{e} <: F t \bar{e}_1$, we check that:

707 $S; \Gamma, f : F.\text{repr } t \bar{e} \vdash F.\text{subcomp } f : F.\text{repr } t \bar{e}_1$

708 Behind the scenes, this typechecking judgment proves the refinement formula in the type of the
 709 $F.\text{subcomp}$ combinator. The following lemma establishes the soundness of $F.\text{subcomp}$:

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

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

717 3.4 Encoding **PURE**, F[★]’s primitive Dijkstra monad

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

722 `let wp a = w:((a → prop) → prop) { ∀p1 p2. (∀x. p1 x → p2 x) → (w p1 → w p2) }`

723 Rather than taking it as primitive, **PURE** can be defined as an indexed effect whose representation
 724 is pure a w, a form of continuation monad where the p:(a → prop) is a “logical continuation” in prop.

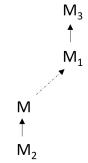
725 `let pure a w = p:(a → prop) → squash (w p) → v:a{p v}`
 726 `let return (x:a) : pure a (λ p → p x) = λ_ _ → x`
 727 `let bind (f:pure a w1) (g:(x:a → pure b (w2 x))) : pure b (λ p → w1 (λ x → w2 x p)) =`
 728 `λp _ → let x = f (λ x → w2 x p) () in g x p () (* run f with a chosen precondition, then run g with p *)`
 729 `let subcomp (w1 w2:wp a) (_:squash (v. w2 p → w1 p)) (f:pure a w1) : pure a w2 = f`
 730 `let if_then_else (w1 w2:wp a) (f:pure a w1) (g:pure a w2) (b:boolean) = pure a (if b then w1 else w2)`
 731 `effect { PURE (a:Type) (w:wp a) with { repr = pure; return; bind; subcomp; if_then_else } }`

736 **PURE** $a : w$ is the type of a pure computation whose result satisfies p , provided $w p$ holds.

737 **3.5 Implementation of indexed effects**

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

741 *Coherence of lifts and effect upper bounds.* When adding a lift M_1 to the signature, our
742 implementation computes all the new lift edges that it induces via transitive closure. For
743 example, if lift M_2 and lift M_3 already exist, this new lift induces a lift M_3 via composition.
744 For all such new edges, if the effects involved already have an edge between them, F^*
745 ignores the new edge and emits a warning. Further, F^* also ensures that for all effects
746 M and M_1 , either they cannot be composed or they have a unique least upper bound.
747 This ensures that the final effect M is unique in the rules T-LET and T-CASE. Finally, F^* ensures
748 that there are no cycles in the effects ordering.



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

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

758 $S; \cdot \vdash F.\text{if_then_else} : a:\text{Type} \rightarrow \overline{x:t} \rightarrow f:F.\text{repr } a \ e_{\text{then}} \rightarrow g:F.\text{repr } a \ e_{\text{else}} \rightarrow b:\text{bool} \rightarrow \text{Type}$
759 and

760 $F.\text{if_then_else} = \lambda a \ \overline{x} \ f \ g \ b. \ F.\text{repr } a \ e_{\text{composed}}$

761 F^* ensures the soundness of the combinator by checking that under the assumption b , the type
762 of f is a subtype (as per $F.\text{subcomp}$) of the composed type, similarly for g under the corresponding
763 assumption *not* b . When the combinator is omitted, F^* chooses a default one that forces the
764 computation type indices for the branches to be provably equal.

765 *Inferring effect indices using higher-order unification.* Our implementation relies on the higher-
766 order unifier of F^* to infer effect indices and arguments of the effect combinators. For example,
767 suppose we have a computation type $F \ t_1 \ \overline{e_1}$ and we want to apply the $\text{lift}_F^{F'}$ combinator, where
768 $\text{lift}_F^{F'} = e$ such that:

769 $S; \cdot \vdash e : a : \text{Type} \rightarrow \overline{x:t} \rightarrow F.\text{repr } a \ \overline{e_f} \rightarrow F'.\text{repr } a \ \overline{e}$

770 To apply this combinator, we create fresh unification variables for the binders a and \overline{x} , and
771 substitute them with the unification variables in $F.\text{repr } a \ \overline{e_f}$ and $F'.\text{repr } a \ \overline{e}$, without unfolding
772 the e_{repr} . We then unify t_1 with the unification variable for a , $\overline{e_1}$ with substituted $\overline{e_f}$, and return
773 (substituted) $F' \ a \ \overline{e}$ as the lifted computation type. This allow us to compute instantiations of the
774 combinators without reifying $F \ t_1 \ \overline{e_1}$ or reflecting the result type of lift. We follow this recipe for all
775 the effect combinators.

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

778 $\text{let bind } a \ b \ t_0 \ t_1 \ (f:\text{gst } a \ t_0) \ (g: a \rightarrow \text{gst } b \ t_1) : \text{gst } b \ (t_0 \oplus t_1) = \dots$

779 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
780 unify the indices of the f argument, i.e. $?u_a$ and $?u_{t_0}$, with the indices of the first computation
781 type int and R. Similarly, for the g argument, $?u_b$ and $?u_{t_1}$ are unified with unit and RW. Finally,

782

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

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

794 4 DIJKSTRA MONADS AND ALGEBRAIC EFFECTS

795 Maillard et al. [2019] present Dijkstra monads as a monad-like structure \mathbf{M} indexed by a separate
 796 monad of specifications \mathbf{W} , forming types of the shape $\mathbf{M} \mathbf{a} \mathbf{w}$, where $\mathbf{w} : \mathbf{W} \mathbf{a}$. In this section, we present
 797 a refined instance of such a construction, a *graded* Dijkstra monad, integrated within a library of
 798 algebraic effects and handlers.

799 Algebraic effects and handlers are a framework for modeling effects in an extensible, composable,
 800 re-interpretable manner, with strong semantic foundations [Plotkin and Power 2003] and several
 801 new languages and libraries emerging to support them [Bauer and Pretnar 2015; Brady 2013; Leijen
 802 2017; Lindley et al. 2017; Plotkin and Pretnar 2009]. We show that algebraic effects and handlers can
 803 be encoded generically using dependent types in F^* , and exposed to programmers as an indexed
 804 effect supporting programming in an abstract, high-level style. Further, we show that operation
 805 labels can be conveniently tracked as an index, much like in existing effect systems for algebraic
 806 effects. Also, we reconcile them with WPs for the particular case of state, proving that functional
 807 specifications can be strengthened from the intensional information of its operations, employing a
 808 unique combination of graded and Dijkstra monads.

809 4.1 A graded monad for algebraic effects

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

```
813 type op = | Read | Write | Raise
  814 let op_inp o = match o with | Read → unit | Write → state | Raise → exn
  815 let op_out o = match o with | Read → state | Write → unit | Raise → empty
  816 type tree0 (a : Type) = | Return : a → tree0 a
  817           | Op : op:op → i:(op_inp op) → k:(op_out op → tree0 a) → tree0 a
```

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

```
824 let ops = list op      let tree (a:Type) (labs:ops) = c:(tree0 a){abides labs c}
  825 where let rec abides (labs:ops) (c : tree0 a) : prop = match c with
  826           | Return _ → T | Op a i k → a ∈ labs ∧ ( ∀ o. abides labs (k o))
```

827 ⁶Our implementation in the supplementary material contains an additional uninterpreted $\text{Other} : \text{int} \rightarrow \text{op}$, and never
 828 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 (_: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 }}
```

4.2 Operations and their handlers, with reflect and reify

844
 845 To add operations to our new effect Alg, the reflect operator is useful, as seen in the generic action
 846 geneff below, which uses reflect to promote a tree to an Alg. Specific instances of operations can
 847 be defined easily using geneff, where for raise we take an extra step of matching on its (empty)
 848 result to make it polymorphic.

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

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

```
854 exception Failure of string
855 let add_st x : Alg int [Read; Raise] = let s = get () in if s < 0 then raise (Failure "error") else x+s
856
 857 When defining effect handlers, one needs access to the tree representation. For instance, the
  858 handle_tree combinator allows all the labs0 operations in c to be handled by h, which in turn may
  859 leave the labs1 operations to be handled, with v the continuation of c's return.
```

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

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

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

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

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

876
 877
 878
 879
 880
 881
 882

883 4.3 AlgWP: A graded Dijkstra monad for stateful Alg programs

884 While Alg above bounds the operations each computation may invoke, there is no way to specify
 885 their order, or any property about the final value and state. To do so, we can bring back the idea of
 886 using a WP calculus to the algebraic setting by adding a new index to the effect. We limit ourselves
 887 to stateful operations and use WPs to specify the behavior according to the state monad: without
 888 fixing an interpretation, it is unclear what can be verified, unless equations are added.

889 Our new effect will refine tree with a stateful WP. We can define a function that takes a
 890 computation tree and computes a “default” stateful WP from it. Then, we take the representation
 891 treewp a 1 wp to be computation trees with operations in 1 whose default WP is pointwise weaker
 892 than its annotated WP (allowing underspecification). This construction is due to Maillard et al.
 893 [2019] but our setting is more general due to the additional grading, not supported in Dijkstra
 894 monads. We begin by defining a predicate transformer monad for stateful programs—st_wp a is the
 895 type of functions transforming postconditions on results and states to preconditions on states.

```
896
897 type post_t (a:Type) = a → state → prop
898 type st_wp (a:Type) = state → post_t a → prop
899 let read_wp : st_wp state = λs₀ p → p s₀ s₀
900 let write_wp : state → st_wp unit = λs _ p → p ()
901 let return_wp (x:a) : st_wp a = λs₀ p → p x s₀
902 let bind_wp (w₁ : st_wp a) (w₂ : a → st_wp b) : st_wp b = λs₀ p → w₁ s₀ p → w₂ s₀ (λ y s₁ → w₂ y s₁ p)
```

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

```
907
908 let rec interp_as_wp #a (t : tree a [Read; Write]) : st_wp a = match t with
909 | Return x → return_wp x
910 | Op Read _ k → bind_wp read_wp (λ s → interp_as_wp (k s))
911 | Op Write s k → bind_wp (write_wp s) (λ () → interp_as_wp (k ()))
912 type rwops = l:ops{l ⊆ [Read; Write]}
913 let treewp (a : Type) (l:rwops) (w: st_wp a) = t:(tree a l){ w ≤ interp_as_wp t }
914 effect { AlgWP (a:Type) (l:rwops) (w:st_wp a) with { repr = treewp; ...}}
915 let soundness (t : unit → AlgWP a l wp) : s₀:state → PURE (a & state) (wp s₀) = ...
```

916 Using this graded Dijkstra monad, we can verify functional correctness properties, which a
 917 graded monad alone cannot capture. For instance, when the state is instantiated to a heap (mapping
 918 locations to values), we can prove that the program below correctly swaps two references, where
 919 AlgPP is simply a pre-/postcondition alias to AlgWP.

```
920 effect AlgPP a l p q = AlgWP a l (λ s₀ post → p s₀ ∧ (forall x s₁. q x s₁ == post x s₁))
921 let swap (l₁ l₂ : loc) : AlgPP unit [Write; Read] (requires λ_ → l₁ ≠ l₂)
922   (ensures λh₀ _ h₁ → h₀ \{l₁; l₂\} == h₁ \{l₁; l₂\} ∧ h₁.[l₁] == h₀.[l₂] ∧ h₁.[l₂] == h₀.[l₁])
923   = let r = !l₂ in l₂ := !l₁; l₁ := r
```

925 More interestingly, the static information in the label index can be exploited by the WP. The
 926 quotient function below strengthens the postcondition of a write-free AlgWP program into addition-
 927 ally ensuring that the state does not change. Operationally, quotient just runs f(), so it can be
 928 seen as a proof that f does not change the state.

```
929
930 val quotient (f : unit → AlgPP a [Read] p q) : AlgPP a [Read] p (λ h₀ x h₁ → q h₀ x h₁ ∧ h₀ = h₁)
```

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

936 937 5 LAYERED INDEXED EFFECTS FOR MESSAGE FORMATTING IN TLS

938 Indexed effects are not just for defining new effect typing disciplines—effect layers stacked upon
 939 *existing* effects can make client programs and proofs more abstract, without any additional runtime
 940 overhead. We demonstrate this at work by stacking two effect layers over EverParse [Ramananandro
 941 et al. 2019], an existing library in F[★] for verified low-level binary message parsing and formatting,
 942 simplifying different aspects of the programs and proofs in each layer.

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

950 951 `effect Stack (a:Type) (pre:mem → prop) (post:mem → a → mem → prop)`

952 Programs in Stack may only allocate on the stack, while reading and writing both the stack and
 953 the heap, with pre- and postconditions referring to mem, a region-based memory encapsulating both
 954 stack and heap. Low[★] provides fine-grained control for general-purpose low-level programming,
 955 at the expense of low-level proof obligations related to spatial and temporal memory safety, and
 956 framing—we aim to simplify these proofs for binary message formatters with domain-specific
 957 abstractions built using indexed effects.

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

960 961 `type example = { left: U32.t; right: U32.t }`

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

964 965 `val example_intro mem (output: buffer U8.t) (offset_from: U32.t) : Lemma`
 966 `(requires valid_from parse_u32 mem output offset_from ∧`
 `valid_from parse_u32 mem output (offset_to parse_u32 mem output offset_from))`
 967 `(ensures valid_from parse_example mem output offset_from)`

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

969 970 `let write_example (output: buffer U8.t) (len: U32.t) (x y: U32.t) : Stack bool`
 971 `(requires λm0 → live m0 output ∧ len == length output) (* memory safety *)`
 972 `(ensures λm0 success m1 → modifies output m0 m1 ∧`
 `(success ==> valid_from parse_example m1 output 0)) (* memory safety & framing *)`
 973 `(* output correctness wrt parser *)`
 974 `= if len < 8 then false (* output buffer too small *)`
 975 `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`

976 977 The user needs to reason about the concrete byte offsets: they need to provide the positions
 978 where values should be written, relying on write_u32 returning the position just past the 32-bit
 979 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

991 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 $f : \text{Write } a$ before p after
 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 p before
 996 parser specification, and ensures that, if successful, it contains binary data valid according to p after
 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 ()
```

1016

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

```
1019
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 specifical parameterized monad evolving an abstract state from
 1023 p before.datatype to p after.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.

1026

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

```

1036 *Using effect subcomp for automatic parser rewriting.* We can embed parser rewriting rules in the
 1037 subcomp combinator for `Write`, as follows:

```

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

```

1040 where `valid_rewrite p2 p2'` is a relation on parser specifications stating that any binary data valid
 1041 for `p2` is also valid for `p2'`. Since the `valid_rewrite` goals can automatically be solved via SMT,
 1042 this allows us to rewrite example as simply: `(write_u32 x; frame (λ_ → write_u32 y))`. The remaining
 1043 overhead is framing; we eliminate it with another indexed effect layered on top of `Write`.

1044 5.2 Automated framing with `FWrite`

1045 Following the frame inference methodology proposed by Fromherz et al. [2021] in the context of a
 1046 concurrent separation logic, we define a new effect `FWrite` that automatically adds frames to the
 1047 computations when sequentially composing them.

```

1048 type frepr (a:Type) pbefore pafter = unit → Write a pbefore pafter
1049 val fbind (a b: Type) (p1 p1' p2 p2': parser)
1050   (frame_f: parser) (frame_g: parser)
1051   (_ : squash (valid_rewrite (parse_pair frame_f p2) (parse_pair frame_g p1'))))
1052   (f: frepr a p1 p2) (g: a → frepr b p1' p2')
1053   : frepr b (parse_pair frame_f p1) (parse_pair frame_g p2')
1054
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
1095 write_vllist_nil starts writing an empty list by writing 0 as its size header. extend_vllist_snoc
1096 assumes that the output buffer contains some variable-sized list immediately followed by an
1097 additional element and “appends” the element into the list by just updating the size header of the
1098 list; thus, the new element is not copied into the list, since it is already there at the right place.
1099 extend_vllist_snoc also dynamically checks whether the size of the resulting list is still within the
1100 bounds expected by the parser, returning an error if not.
```

1101 The data format specified by `parse_vllist p min max` and implemented by those `FWrite` combina-
 1102 tors corresponds to the format of variable-sized lists prefixed by their byte size as mandated by the
 1103 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 refering the reader to the Steel and DY* papers
 1125 for more details.

1126

1128

6.1 Steel

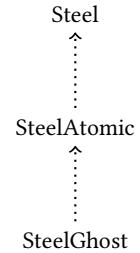
1129 Steel is a language for developing and proving concurrent, dependently-typed F^* programs. Steel's
 1130 program logic is based on a shallow embedding of concurrent separation logic in F^* , while also
 1131 enabling Hoare-style reasoning using a variant of heap predicates, called *selector predicates*.

1132 To enable smooth interoperation between separation logic and Hoare logic, Steel encodes pre-
 1133 and postconditions as effect indices; we show a simplified version of the Steel effect representation:

```
1134
1135 let selpre fp = rheap fp → prop      let selpost fp a fp' = rheap fp → a → rheap fp' → prop
1136 val steel_repr (a:Type) (fp:słprop) (fp':a → słprop) (req:selpre fp) (ens:selpost fp a fp') : Type
```

1137 Separation logic specifications rely on assertions of type $słprop$, and are encoded through a
 1138 precondition fp , and a return value dependent postcondition fp' . Similarly to the Hoare monad
 1139 from §2.3, selector specifications consist of a precondition req and a 2-state postcondition ens .
 1140 Note that req and ens operate on states (i.e., heaps) that are parameterized by the separation logic
 1141 specifications: $rheap fp$ is a restricted heap corresponding exactly to the predicate fp . This is another
 1142 instance of a hybrid, dependent indexing structure, where the fp indices constrain the selector
 1143 predicates.

1144 To reason about concurrent programs, Steel also models atomic computations,
 1145 as well as a notion of ghost state, which can be manipulated through
 1146 ghost, computationally irrelevant computations. Ghost and atomic computations
 1147 are separated from generic Steel functions, and are thus modeled
 1148 as their own effects, `SteelGhost` and `SteelAtomic`, with two additional effect
 1149 indices to encode verification conditions related to atomicity. Nevertheless,
 1150 a ghost computation can always be seen as atomic, while an atomic computation
 1151 is but a special case of a generic Steel computation. Steel captures this
 1152 hierarchy through lifts between its different effects, which are automatically
 1153 inserted by our framework when needed.



1154 In Steel, indexed effects thus provide a foundation to structure reasoning, enabling, for instance,
 1155 a separation of verification conditions related to atomicity and separation logic. Leveraging this
 1156 structure, Steel automates framing reasoning, using a methodology similar in spirit to the one
 1157 presented in §5.2, albeit applied to a full-fledged, impredicative, concurrent separation logic, which
 1158 simplified the development of a wide variety of verified libraries, ranging from self-balancing trees
 1159 and concurrent queues to 2-party session types.

1160

6.2 DY*

1161 DY* is an F^* -based framework for symbolic verification of security protocols and has been used
 1162 for the first symbolic analysis of the Signal protocol, the messaging protocol used in WhatsApp,
 1163 while accounting for an unbounded number of ratcheting rounds.

1164 Protocols sessions in DY* are modeled as partial, stateful F^* functions that may raise exceptions
 1165 and the underlying state is a global, monotonic trace that tracks the interleaved execution of
 1166 sessions. This is represented as an indexed effect for a state and exception Dijkstra monad, called
 1167 `Crypto`. All the security protocols in DY*, including Signal, are written in the `Crypto` effect, and
 1168 verified against the trace-based properties expressed as specifications in the Dijkstra monad.

```
1169
1170
1171 type wp (a:Type) = (option a → trace → prop) → trace → prop
1172 (* The monotonicity property of the trace is internalized in the repr via extends *)
1173 type crypto_repr (a:Type) (w:wp a) =
1174   s0:trace → PURE (option a & trace) (λ p → w (λ x s1 → s1 `extends` s0 ==> p (x, s1)) s0)
```

1177 Without indexed effects, defining such an effect in F^* would not be possible, and using an
 1178 axiomatized effect for verifying security properties is clearly suboptimal. Though one could derive
 1179 an effect for state and exceptions using the Dijkstra Monads for Free methodology [Ahman et al.
 1180 2017], it does not allow internalizing trace monotonicity, as they have done here.

1181 **7 RELATED WORK & CONCLUSIONS**

1183 We have discussed several strands of related work throughout the paper. We focus here on relating
 1184 our work to two main themes not discussed in detail elsewhere.

1185 *Unifying frameworks for effectful programming and semantics.* Given the variety of monad-like
 1186 frameworks for effects, a unifying theory that accounts for all the variants is a subject of some
 1187 interest. Filinski [1999] presents a framework for specifying and implementing layered monads,
 1188 focusing on implementing them uniformly with delimited continuations and state, though does
 1189 not consider indexed monads. Tate's (2013) productors and, equivalently, Hicks et al.'s (2014)
 1190 polymonads are attempts at subsuming frameworks, Tate focusing more on the semantics while
 1191 Hicks et al. consider programmability, though neither handle dependently typed programs. Bracker
 1192 and Nilsson [2015] provide a Haskell plugin for polymonad programming—our implementation
 1193 also provides support for polymonads, where not only the indices but also the effect label can vary
 1194 among the computation arguments to bind, a feature used in Steel [Fromherz et al. 2021]. Orchard
 1195 et al. [2020] propose to unify graded and parameterized monads by moving to category-indexed
 1196 monads, studying them from a semantic perspective only, while also working in a simply typed
 1197 setting. Orchard and Petricek [2014] also propose a library to encode effect systems with graded
 1198 monads in Haskell.

1199 *Programming and proving with algebraic effects.* Our library for algebraic effects is perhaps
 1200 related most closely to Brady's (2013) Effects DSL in the dependently typed language Idris. The
 1201 main points of difference likely stem from what is considered idiomatic in Idris versus F^* . For
 1202 instance, the core construct in the Idris DSL is a type indexed by a list of effects (similar to our
 1203 tree a1)—whereas in Idris the indexing is intrinsic, our trees are indexed extrinsically with a
 1204 refinement type, enabling a natural notion of subsumption on indices based on SMT-automated
 1205 effect inclusion. Idris' core effects language is actually a parameterized monad—our supplement
 1206 and full version of the paper show a similarly parameterized version of our tree type. By packaging
 1207 our trees into an effect, we benefit from automatic elaboration, avoiding the need for monadic
 1208 syntax, idiom brackets and the like, with implicit subsumption handled by SMT. Further, unlike
 1209 Brady, we provide a way to interpret Read-Write trees into a Dijkstra monad, enabling functional
 1210 correctness proofs. While we have only taken initial steps in this direction, we appear to be the
 1211 first to actually verify stateful programs in this style. Maillard et al. [2019] propose a tentative
 1212 semantics to interpret algebraic effect handlers with Dijkstra monads, and use their approach to
 1213 extrinsically verify the totality of a Fibonacci program with general recursion. Our work builds on
 1214 theirs, requires fixing the interpretation of the operations, but yields a methodology to do proofs of
 1215 stateful programs. Besides, with indexed effects, we get to choose whether to work with Dijkstra
 1216 monads or not—in contrast, Maillard et al.'s framework cannot support the list-of-effects indexed
 1217 graded monad. Algebraic effects have also been embedded in Haskell in several styles, notably
 1218 by Kiselyov and Ishii's (2015) “freer” monads, relying on encodings of dependent types in Haskell's
 1219 type system to also index by a list of effect labels, while focusing also on efficient execution, a topic
 1220 we have not yet addressed for our Alg effect.

1221 *Conclusions.* Embracing the diversity of indexed monad-like constructions, and aiming to benefit
 1222 from them when programming with effects in a dependently typed language, we have designed
 1223

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 REFERENCES

1234 Danel Ahman, Cédric Fournet, Cătălin Hrițcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness:
 1235 Foundations and Applications of Monotonic State. *PACMPL 2, POPL* (jan 2018), 65:1–65:30. <https://arxiv.org/abs/1707.02466>

1236 Danel Ahman, Cătălin Hrițcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and
 1237 Nikhil Swamy. 2017. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages
 (POPL)*. ACM, 515–529. <https://doi.org/10.1145/3009837.3009878>

1238 Robert Atkey. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19 (2009), 335–376. Issue 3-4.
 1239 <https://doi.org/10.1017/S095679680900728X>

1240 Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*
 1241 84, 1 (2015), 108–123. <https://doi.org/10.1016/j.jlamp.2014.02.001>

1242 Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations. In *Programming
 1243 Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings (Lec-
 1244 ture Notes in Computer Science, Vol. 4279)*, Naoki Kobayashi (Ed.). Springer, 114–130. https://doi.org/10.1007/11924661_7

1245 Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim
 1246 Würtele. 2021. DY*: Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. (2021).
 1247 <https://publ.sec.uni-stuttgart.de/bhargavanbichhavatdohosseynikuesterschmitzwuertele-eurosp-2021.pdf> To appear.

1248 Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Cătălin Hrițcu, Samin
 1249 Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko,
 1250 Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-
 1251 Béguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd
 1252 Summit on Advances in Programming Languages*. [http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-
 1254 2017-1.pdf](http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-

 1253 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 Ramananandro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. In *Proceedings of
 1269 the International Conference on Functional Programming (ICFP)*.

1270 Michael Hicks, Gavin M. Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. 2014. Polymonadic Programming. In
 1271 *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12
 1272 April 2014 (EPTCS, Vol. 153)*, Paul Levy and Neel Krishnaswami (Eds.). 79–99. <https://doi.org/10.4204/EPTCS.153.7>

1273 Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In
 1274 *Proceedings of the 14th International Conference on Formal Methods* (Hamilton, Canada) (FM'06). Springer-Verlag, Berlin,
 1275 Heidelberg, 268–283. https://doi.org/10.1007/11813040_19

1276 Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *The 41st Annual ACM SIGPLAN-
 1277 SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh
 1278 Jagannathan and Peter Sewell (Eds.). ACM, 633–646. <https://doi.org/10.1145/2535838.2535846>

1279

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 Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. <http://dl.acm.org/citation.cfm?id=3009872>

1278 1279 Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.

1280 1281 Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (Paris, France) (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 500–514. <https://doi.org/10.1145/3009837.3009897>

1282 1283 1284 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hrițcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (July 2019), 29 pages. <https://doi.org/10.1145/3341708>

1285 1286 1287 Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hrițcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP)*. Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

1288 1289 1290 Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5–8, 1989*. IEEE Computer Society, 14–23. <https://doi.org/10.1109/LICS.1989.39155>

1291 1292 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5–6 (2008), 865–911. <http://ynot.cs.harvard.edu/papers/jfpsep07.pdf>

1293 1294 1295 Dominic Orchard, Philip Wadler, and Harley Eades III. 2020. Unifying graded and parameterised monads. In *Proceedings Eighth Workshop on Mathematically Structured Functional Programming, MSFP at ETAPS 2020, Dublin, Ireland, 25th April 2020 (EPTCS, Vol. 317)*, Max S. New and Sam Lindley (Eds.), 18–38. <https://doi.org/10.4204/EPTCS.317.2>

1296 1297 1298 Dominic A. Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4–5, 2014*, Wouter Swierstra (Ed.). ACM, 13–24. <https://doi.org/10.1145/2633357.2633368>

1299 1300 1301 1302 Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. <https://doi.org/10.1023/A:1023064908962>

1303 1304 1305 Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

1306 1307 1308 Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hrițcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. <https://doi.org/10.1145/3110261>

1309 1310 1311 1312 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the 28th USENIX Conference on Security Symposium (Santa Clara, CA, USA) (USENIX Security 2019)*. USENIX Association, USA, 1465–1482.

1313 1314 1315 E. Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446. <https://tools.ietf.org/html/rfc8446>

1316 1317 1318 Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. 2011. Lightweight monadic programming in ML. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011 (ICFP '11)*, 15–27. <https://www.cs.umd.edu/~mwh/papers/swamy11monad.html>

1319 1320 1321 Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. <https://www.fstar-lang.org/papers/mumon/>

1322 1323 Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.* 4, ICFP, Article 121 (Aug. 2020), 30 pages. <https://doi.org/10.1145/3409003>

1324 1325 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design 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-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) (POPL '13). Association for Computing Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/2429069.2429074>

1325 Philip Wadler. 1992. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 1–14. <https://doi.org/10.1145/143165.143169>

1329

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