Composing and Staging Effect Handlers

ZHIXUAN YANG, University of Exeter, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom

This paper presents the design and engineering of an expressive and efficient library for handlers of higher-order effects in Haskell. The main novelty of this library is a toolbox of *handler combinators*, which (1) capture patterns of interaction of handlers and at the same time (2) fuse iterative rounds of handling into a single-round of handling. Compared to hand-rolled monadic code, the library imposes only amortised $O(1)$ runtime overhead for both invoking an operation and installing a handler for an operation, while providing the extra expressivity and modularity benefits of effect handlers.

To further reduce the constant-factor overhead, the library additionally supports two styles of meta-programming: (1) *light staging*, which computes the handler combinators at compile time, but operation calls are not staged; (2) *full staging*, which computes both handlers and operation calls at compile time as much as possible. Light staging improves performance with very little change to the programming interface, whereas full staging necessitates a bigger change of the programming interface but is able to generate highly optimised code that in some cases can be faster than hand-rolled code.

Additional Key Words and Phrases: effect handlers, staging, meta-programming, functional programming

1 INTRODUCTION

From a practical perspective, *effect handlers*, introduced by Plotkin and Pretnar [2009, 2013] and further developed by many others, are a programming language construct providing native support for extending the programming language with new syntax (such as nondeterministic choices, mutable state, and coroutines) and implementing the semantics (in possibly more than one way), all *without leaving the language*. Therefore effect handlers may be seen as a form of *language-oriented programming* (LOP) [Felleisen et al. 2018; Ward 1994]. Compared to other forms of LOP, effect handlers stand out for their relatively low cognitive overhead: imperative programmers may view effect calls as *resumable exceptions*, while functional programmers may view effect handlers as a native form of *syntax tree folds*. In both cases, the programmer has a simple mental model of effect handlers, while enjoying the expressivity brought by them. We believe that this is an important reason for the rapid adoption of effect handlers in many programming languages.

Since Plotkin and Pretnar's early proposal, the large body of research on effect handlers may be loosely divided into three categories in terms of their focuses:

- *Efficiency*. Naive implementations of effect handlers (both in standalone languages or as programming libraries in existing languages) usually incur non-trivial runtime overhead, so efficient ways of implementing effect handlers are studied by, to just name a few, Karachalias et al. [2021]; Kiselyov and Ishii [2015]; Müller et al. [2023]; Schuster et al. [2020]; Trifanov and Schrijvers [2026]; Wu and Schrijvers [2015]; Xie and Leijen [2021].
- *Expressivity*. The simplicity of effect handlers as a form of language-oriented programming comes from the fact that effect handlers impose a strong restriction on what kind of syntax the programmer can extend the language with — the ones known as *algebraic operations* [Plotkin and Power 2003], which are equivalent to *generic operations* $o(p) : A$, those accepting a parameter p of some type P and returning a result of some type A. The line of research on *higher-order algebraic effects* and their handlers precisely aims to alleviate this restriction, allowing effectful operations to delimit scopes around their arguments (which essentially means that arguments to operations can be computations instead of just values)

Authors' addresses: Zhixuan Yang, University of Exeter, Exeter, United Kingdom, z.yang2@exeter.ac.uk; Nicolas Wu, Imperial College London, London, United Kingdom, n.wu@imperial.ac.uk.

[Bach Poulsen and van der Rest 2023; Bosman et al. 2024; Frumin et al. 2024; Piróg et al. 2018; van den Berg and Schrijvers 2024; Wu et al. 2014; Yang and Wu 2023, 2026].

- *Ease of reasoning*. Similar to other powerful control operators, when effect handlers are used pervasively, it can be a big cognitive load for the programmer to track what effects a program may invoke. To alleviate this problem, *type-and-effect systems* are studied, which statically track the effects that may be caused by a program [Bauer and Pretnar 2014; Leijen 2017; Lindley et al. 2017; Tang and Lindley 2026; Tang et al. 2025; van Rooij and Krebbers 2025] and possibly even the algebraic laws on the operations assumed by the program [Kammar and Plotkin 2012; Lukšič and Pretnar 2020].

The dissection above is by no means absolute, as these three goals are almost always interconnected; for example, precise type-and-effect systems usually enable efficient implementations, while higher-order effects pose new challenges to efficient implementations. In this paper we report how these three design goals are reconciled in the design and implementation of Effective, a new library for handling higher-order effects in the language Haskell.

Our technical contributions in the design and implementation of Effective are as follows:

(1) We present a novel domain-specific language for composing handlers of *higher-order* effects in the form of a library of handler combinators. These combinators not only allow the programmer to express intricate interactions of effects, but also reduce the runtime overhead of effect handlers by representing effect handlers using efficient data structures. Notably, we achieve amortised $O(1)$ runtime overhead for both invoking an effectful operation and installing a handler, compared to directly writing monadic programs without any abstractions such as effect handlers or mtl-style typeclasses.
(2) We show how the runtime performance of effect handlers can be further improved by *staged programming*. We propose two complementary approaches to staged effectful programming: *light staging* and *full staging*. Light staging evaluates our handler combinators at compile time, while effectful programs are still non-staged. Full staging performs effect handling solely at compile time, which generates efficient low-level monadic code that does not use effect handlers or comparable mechanisms at all. Light staging is a relatively simple technique and imposes almost no changes to the programming interface, whereas full staging is more intricate, requiring the programmer to slightly change how effectful programs are written and sometimes explicitly use certain operations for controlling code generation.

The paper is organised as follows: in Section 2 we provide an extended high-level overview of the problems that we try to solve and our solutions, using a running example of CCS-style concurrency. In Section 3, we present the foundational layer of our library – how operations and effectful programs are represented. In Section 4, we show the first part of our contributions – algebra transformers, handlers, and their combinators. In Section 5, we present the second part of our contributions – improving performance of effect handlers by staging. In Section 6, we conclude by discussing the performance, limitation, and related work of our library.

2 COMPOSING AND STAGING HANDLERS: AN OVERVIEW

Before going into any technical details, we first show a high-level overview of the problems in making effect handlers efficient and expressive and our main ideas to address them: *handler fusion* (Section 2.1), *handler wiring diagrams* (Section 2.2), *higher-order effects* (Section 2.3), *light staging* (Section 2.5), *full staging* (Section 2.6). All these ideas are demonstrated concretely by a running example of handling a program with concurrency and outputting (Section 2.4).

2.1 Handler Fusion

To discuss efficiency issues of effect handlers, let us first recap why naive implementations of effect handlers are slow. In a simple but reasonably realistic operational-semantics-based implementation of effect handlers, such as the generalised CEK machine for effect handlers described by Hillerström and Lindley [2016], the program **handle** p **with** h (handling the effects in the program p with the handler h) roughly works by pushing the handler h onto a stack of currently activated handlers, and then continuing as p. Later when an effectful operation o inside p is triggered, the machine goes over the stack of activated handlers, and searches for the first handler h' that handles o, and then the control flow jumps to the handling code of h' for o, with the current continuation captured up to the boundary delimited by h'.

A noticeable source of inefficiency is the process of searching for the first handler of an operation in the stack of activated handlers. Suppose we have deeply nested layers of n handlers h_i:

$$\textbf{handle}\ \{\cdots\ \{\textbf{handle}\ \{\textbf{handle}\ p\ \textbf{with}\ h_1\}\ \textbf{with}\ h_2\}\ \cdots\}\ \textbf{with}\ h_n \tag{1}$$

where each handler h_i handles exactly an operation o_i. Then in the program p, every time the operation o_n is invoked, we need to go through the whole n layers of handlers on the stack to find h_n, although we could have statically known o_n is handled by h_n. Suppose that the program p consists of exactly n calls to $o_n : () \to ()$ and the handler h_n handles o_n in constant time then the program (1) has size $O(n)$ but runs in $O(n^2)$-time.

This problem does not just arise in operational-semantics-based implementations. In implementations of effect handlers based on denotational semantics, effectful programs are represented using variants of *free monads* [Kammar et al. 2013; Kiselyov and Ishii 2015], which are conceptually trees whose nodes are effectful operations, and subtrees of a node are the continuations of the operation. A handler is then a *fold* in the parlance of functional programming that traverses trees of operations and replaces the operations it handles with the corresponding semantic implementation (typically called an *algebra* in this context). For the program (1) above, p is a chain of length n, and all handlers $h_1, h_2, \cdots, h_n$ will traverse this chain of length n (the handlers other than h_n will find no operations in p that they want to handle). Hence the running cost is still $O(n^2)$.

The operational implementation and the denotational implementations described above are two sides of the same coin. In the operational view, programs are conceptually the acting subject that searches for the corresponding handlers. In the denotational view, programs now are static objects (syntax trees), and handlers become the acting subject that processes data. Borrowing Paul Levy's famous *is-does distinction* from call-by-push-value [Levy 2003],

the operational view of effect handling:	an operation *does*,	a handler *is*;
the denotational view of effect handling:	an operation *is*,	a handler *does*.

But the running cost associated to each *operation-handler pair* is the same in these two dual views. In the present paper, we mainly take the denotational view, as it is the more natural perspective when implementing effect handlers in a functional programming language.

To reduce the inefficiency of nested handlers in situations like (1), our first main idea is:

Idea 1. By defining a *handler fusion* combinator $h_1 \triangleright h_2$ such that for all programs p

$$\textbf{handle}\ \{\textbf{handle}\ p\ \textbf{with}\ h_1\}\ \textbf{with}\ h_2\ =\ \textbf{handle}\ p\ \textbf{with}\ (h_1 \triangleright h_2), \tag{2}$$

we can turn nested handlers (1) into a single handler $(h_1 \triangleright h_2 \triangleright \cdots \triangleright h_n)$, whose operation clauses can then be efficiently stored in a *random-access array*. When an effectful program invokes an operation o_i, the corresponding handling code can then be found in $O(1)$-time.

A rough intuition for handle fusion $h_1 \triangleright h_2$ is that it takes the union of the clauses of the two handlers. But there are two caveats: (1) we have to be careful about the interaction between these two handlers, since the inner handler h_1 itself can also invoke operations that should be handled by the outer handler h_2. For example, supposing $o_1, o_2 : () \to ()$, then

$$\textbf{handle}\ \{\textbf{handle}\ p\ \textbf{with}\ \{\, o_1\ \_\ k \mapsto o_2\ ();k\ ()\,\}\}\ \textbf{with}\ \{\, o_2\ \_\ k \mapsto body_2\,\} \tag{3}$$

is *not* the same as handling with the simple union of the two handlers

$$\textbf{handle}\ p\ \textbf{with}\ \{\, o_1\ \_\ k \mapsto \{\, o_2\ ();k\ ()\,\};\quad o_2\ \_\ k \mapsto \{\, body_2\,\}\,\}$$

because the operation call o_2 in the handler of o_1 should also be handled as $body_2$ according to the standard semantics of effect handlers. (2) Arrays support efficient indexing but not efficient appending, which we need when fusing two handlers together. So we use instead *finger trees* (which support log-time appending and indexing) to store handler clauses when we are building handlers, and only when we apply a handler to a program, we build an array from the finger tree.

Therefore to correctly define $h_1 \triangleright h_2$ that satisfies the property (2), we need a type-and-effect system that tracks what effects a handler *consumes* and *produces*. For example in (3), the type for the inner handler in our library will record the fact that it consumes o_1 and produces o_2. Therefore when fusing the two handlers in (3), we know that we need to replace the operation call o_2 (together with its continuation k ()) when handling o_1 with the corresponding handler of o_2:

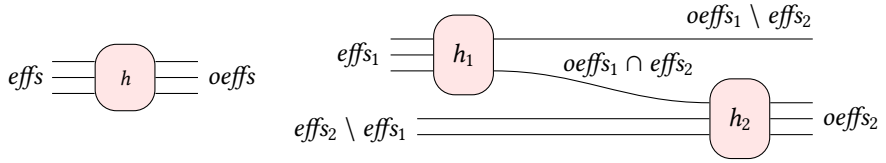
$$(3)\ =\ \textbf{handle}\ p\ \textbf{with}\ \{\, o_1\ \_\ k \mapsto \{\ body_2\ \};\quad o_2\ \_\ k \mapsto \{\, body_2\,\}\,\}$$

In this example, for the sake of demonstration we have made the handling clause of o_1 invoke the continuation k () immediately after the operation call to o_2, so that we can simply replace $\{\, o_2\ ();k\ ()\,\}$ with $body_2$. In general we do not have this restriction, and we need to substitute the continuation parameter k appropriately when inlining $body_2$ in the handling clause of o_1.

As the name suggests, handler fusion $h_1 \triangleright h_2$ is essentially an instance of *fold fusion* [Hinze et al. 2011] in functional programming, which has already been used to optimising [Wu and Schrijvers 2015] and reasoning about [Yang and Wu 2021] effect handlers before. Compared to the previous work, what is new is that we *explicitly* compute fused handlers using efficient data structures, instead of unreliably relying on compiler optimisations to do fusion for us.

2.2 Handler Wiring Diagrams

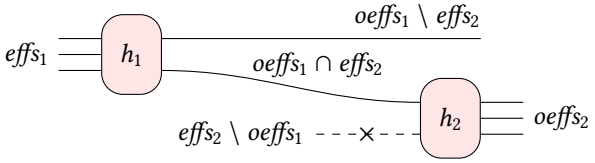
We can visualise a handler h consuming effects $effs$ (the *'input effects'* of h) and producing effects $oeffs$ (the *'output effects'* of h) as a box with wires on the left:



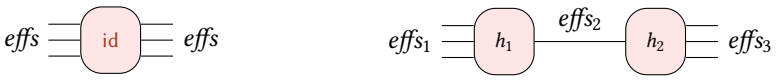
More than one wire is drawn for $effs$ and $oeffs$ to indicate that $effs$ and $oeffs$ are *sets* of effectful operations. Then the fusion combinator $h_1 \triangleright h_2$ can be visualised as the diagram above on the right. The resulting fused handler has input effects $effs_1 \cup (effs_2 \setminus effs_1)$, which is exactly $effs_1 \cup effs_2$. Moreover, when an operation is in the intersection $effs_1 \cap effs_2$, from the diagram we see that it is handled by the handler h_1. The output effects of the fused handler is $(oeffs_1 \setminus effs_2) \cup oeffs_2$, which means that any operation produced by h_1 is also handled by h_2. The flow of operations depicted in this diagram is exactly the standard semantics for $\textbf{handle}\ \{\textbf{handle}\ \cdots\ \textbf{with}\ h_1\}\ \textbf{with}\ h_2$.

The diagram of fusion suggests our next main idea:

Idea 2. We can design a domain-specific language for composing handlers. For example, besides the fusion combinator above, we can also use h_2 to handle the operations produced by h_1 but not any 'upstream operations', as shown in the diagram below. The dashed wire $effs_2 \setminus oeffs_1$ indicates the capability of h_2 to handle these effects is now ignored. In Effective this combinator is called *piping* h_1 and h_2 and denoted by $h_1 \setminus\setminus h_2$.



Readers who like category theory might interject: it looks like a category! Indeed, we can treat handlers as morphisms from input effects to output effects, and we have identity handlers id that 'handle' operations by simply re-produce them and sequential handler composition:



(4)

In this paper, we will not develop the theory of this category formally, but we will use ideas of category theory informally to organise our language of handler combinators.

2.3 Higher-Order Effects

Handler combinators of Effective form an expressive domain-specific language for composing effect handlers. Although they are already very useful in the setting of first-order effects, their power truly shines when they are used with *higher-order* computational effects. First-order effects are operations that accept/return *values* as their arguments/output, while higher-order effects are operations that accept/return *computations* as their arguments/output. For instance, *parallel composition* $p \parallel q$ in concurrent programming is higher-order, because the arguments p and q are not simple values but computations. In the same vein, the operation **try** p **catch** h that catches exceptions produced by the computation p using the computation h is also a higher-order operation.

Operations like $p \parallel q$ and **try** p **catch** h are also called *scoped operations* [Piróg et al. 2018], because the intuition is that they are not 'atomic' but are operations acting on certain scopes of computations. Following the common syntax of using braces to mark scopes, we would then write $\{\cdots\} \parallel \{\cdots\}$ and **try** $\{\cdots\}$ **catch** $\{\cdots\}$. Scoped operations are the most common form of higher-order operations, but there are also higher-order operations that are more complex than just delimiting scopes, for example, the operator **shift**$(k.\ p)$ from shift/reset delimited control not only delimits a scope around the argument p but also binds a variable k in this scope.

In calculi with handlers of first-order effects, we can use handlers to implement scoped operations. Indeed, **try** p **catch** h was exactly the motivating example that Plotkin and Pretnar [2009] used to introduce effect handlers. However, by making scoped operations operations in their own right, they enjoy the benefits of the approach of effect handlers in the same way as first-order operations. This is the main motivation for the research on higher-order effects.

In Effective, we follow the theoretic foundation developed by Yang and Wu [2023, 2026] to implement higher-order effects. A prominent feature in Yang and Wu's proposal is that they advocate for treating *value returning* and *sequential composition* as operations of every computational effect. Therefore every handler should also give semantics to these two operations (i.e. every handler must come with a monad). Yang and Wu's [2026] argument for this choice was that returning and sequential composition are two fundamental operations in every notion of computation.

2.4 An Example with Concurrency

Before explaining how we further improve the performance of Effective by meta-programming, let us sketch a concrete example to demonstrate the ideas so far. Suppose we are interested in the effects of concurrency in the style of Milner's [1980] *calculus of communicating systems* (CCS) together with an effect for outputting. The effect of CCS is parameterised by a type a of actions, and we have three operations: performing an action or a co-action, restricting an action in a scope, and running two processes in parallel. The effect of outputting will have just one operation, parameterised by the type of output w. In Effective, these operations can be defined by writing[1]

```
$(makeGen [|act :: ∀a. Either a a → ()|])      $(makeScp [|par :: 2|])
$(makeScp [|res :: ∀a. a → 1|])                $(makeGen [|tell :: ∀w. w → ()|])
```

which uses the Template Haskell helper functions makeGen and makeScp provided by Effective to generate the boilerplate for generic operations and scoped operations, such as the type, the pattern synonym, and the term-level functions for each operation. We emphasize that the arguments to makeGen and makeScp are not really the type of these operations in Haskell, but just a convenient syntax encoding information for makeGen and makeScp to generate boilerplate code. The real Haskell types of the operations generated by makeGen and makeScp are

```
res :: (Res a ∈ effs) ⇒ a → Prog effs a → Prog effs a
par :: (Par ∈ effs) ⇒ Prog effs a → Prog effs a → Prog effs a
act :: (Act ∈ effs) ⇒ Either a a → Prog effs ()     tell :: (Tell w ∈ effs) ⇒ w → Prog effs ()
```

where Prog *effs a* is the type of programs with effects *effs* and return type a.

In this example, we will use a type **data** HS = Handshake | Stdout with exactly two constructors as actions (the parameter a for act and res) and String as the type of output (the parameter w for tell). Using the operations that we have just declared, we can write a program:

```
br :: ([Act HS, Res HS, Par, Tell String] ⊆ effs) ⇒ Prog effs ()
br = par (res Handshake $ par (do tell "I am just a poor boy"; act (Left Handshake))
                              (do act (Right Handshake); tell "I need no sympathy"))
         (tell "Oh poor boy")
```

This program uses par to run three processes in parallel, each outputting a string. The first two processes are synchronized by an action Handshake – recall that in CCS, every action is synchronised with its co-action, which we represent using the Either type. Additionally, the action Handshake is restricted within the scope of these two processes. The operation of restriction res a p in CCS is like installing a network firewall preventing all network traffic about action a from going into and out of the process p. So if we compose br with another program p, the handshaking action of br will not be confused with the handshaking actions of p.

Effective comes with a box of handlers for common effects such as CCS-style concurrency and outputting. For example, there are handlers for concurrency and outputting based on their standard denotational models, known as the *resumption monad transformer* and *writer monad transformer* respectively [Moggi 1989a]. However, in this introduction we shall play with some more interactive ways of handling them using the native IO monad of Haskell. To begin with, we can handle outputting by translating the Tell operation to putStr :: String → IO () of Haskell:

```
tellIO :: Handler [Tell String] [Alg IO] [ ] a a
tellIO = interpret1 $ λ (Tell w k) → do io (putStr w); return k
```

[1] Our font convention is that variables are in *serif* and defined constants are in sans serif and are usually clickable links.

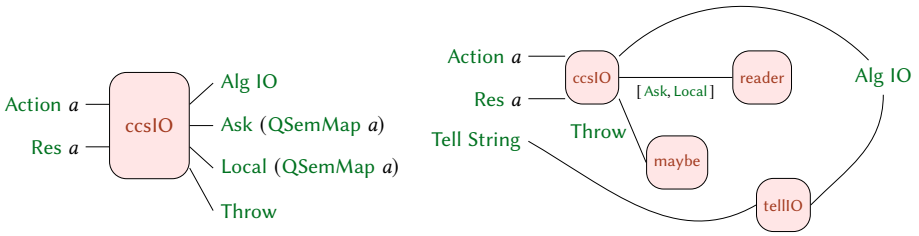


Fig. 1. Wire diagrams for ccsIO and ccsTell

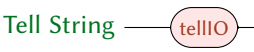
The type of this handler tells us that it consumes exactly one operation Tell String and produces an operation Alg IO, which is the effect in Effective for Haskell-native IO. The way to use the operation Alg IO is the function io :: (Alg IO ∈ *effs*) ⇒ IO *a* → Prog *effs a*. The third type parameter of Handler above is the empty list [ ], which means that the underlying carrier transformer is the trivial one – we can ignore this parameter for now in this section. The last two type parameters of Handler are the types of return values before and after applying this handler. In this case they are the same (polymorphic) type *a*, which means that this handler can work with any return-value type and the handler does not change the type of return values.

The definition of tellIO uses the handler combinator interpret1 from the library, which constructs a handler from a function that maps an operation call *eff m x* to an effectful program Prog *oeffs x*:

$$\text{interpret1} :: \text{HFunctor } \mathit{eff} \Rightarrow (\forall m\ x.\ \mathit{eff}\ m\ x \to \text{Prog } \mathit{oeffs}\ x) \to \text{Handler } [\mathit{eff}\,]\ \mathit{oeffs}\ [\,]\ a\ a$$

From this signature we also notice that in Effective, signatures of (higher-order) effects are internally represented as 'higher-order functors'. We will come back to this in Section 3.

In our handler-wiring diagrams, the handler tellIO can be drawn as



For visual clarity we only track the first two parameters of Handler [Tell String] [Alg IO] [ ] *a a* in the diagram, although the other parameters also matter in handler composition. For example, the type of the sequential composition combinator in the right of (4) is

$$\text{Handler } \mathit{effs}_1\ \mathit{effs}_2\ \mathit{ts}_1\ a\ b \to \text{Handler } \mathit{effs}_2\ \mathit{effs}_3\ \mathit{ts}_2\ b\ c \to \text{Handler } \mathit{effs}_1\ \mathit{effs}_3\ (\mathit{ts}_1 +\!\!+ \mathit{ts}_2)\ a\ c$$

Next we define a handler for CCS-style concurrency using the *quantity semaphore* primitive of Haskell [Peyton Jones et al. 1996]. The idea is to use a pair (s_1, s_2) of binary semaphores to implement the synchronisation on a CCS action *a*. Performing the action *a* is translated to the native IO-effect **do** {waitQSem s_1; signalQSem s_2}, while performing its co-action $\overline{a}$ is translation to the IO-effect **do** {signalQSem s_1; waitQSem s_2}. In this way, no matter whether the action *a* or the co-action $\overline{a}$ is performed first, it is guaranteed that the processes are blocked until they synchronise. To maintain the mapping from the CCS-actions to the semaphores, the handler will use the *reader effect* storing a hash map, and the scoped operation of restriction res *a p* is implemented as creating a new pair of semaphores and locally binds *a* to this pair within *p*. The handler also produces the effect Throw when an action is performed before the corresponding semaphores are created (using the operation res). The reader can find the full code in the supplemented code, and the diagram of the handler ccsIO is the left one in Figure 1.

Now we can use the previously mentioned fusion (▷) and piping (\\\) combinators to chain together tellIO, ccsIO, and the standard handlers for the reader and exception effects:

```
ccsTell = (ccsIO \\ maybe \\ reader empty) ▷ tellIO
```

The diagram of this handler is the right one in Figure 1. Applying this handler to our effectful program br is achieved by ex_1 = handleIO′ ioPar ccsTell br where ex_1 :: IO (Maybe ()). The function handleIO′ applies the handler ccsTell to the program br. Recall that br also uses parallel composition Par, but Par is not handled by ccsTell, so handleIO′ takes in another argument ioPar, which is the implementation of Par on the IO monad (using the primitive forkIO function).

If we run ex_1 a few times, we see that the output looks like `OhI paomo rj ubsoty...` and in fact is random. This is because the tell operations of different processes are run in parallel. Now suppose that we would like the tell operations from different processes to be atomic. We can implement this behaviour by defining a new handler that translates every tell operation into three operations: acquiring a specific lock for outputting, re-producing the tell operation, and releasing the lock. Locks in CCS can be implemented by having a special *daemon process* that repetitively performs coactions $\overline{a}$ and actions a. Other processes then acquire the lock by action a and release the lock by coaction $\overline{a}$. We will use Stdout :: HS as the action for this purpose.

However, one more problem is that who should kick-start the daemon process? One solution is to define a special handleCCS p = handleIO′ _ (par daemon p) but this undermines the modularity of effect handlers. For situations like this, an element of Handler is defined to have two components: the algebra of the operations and a *runner* that can do some initialisation or finalisation work.

```
lock :: Handler [Tell String] [Tell String, Act HS, Par, Res HS] [] a a
lock = Handler (Runner $ λ oalg p →
    let daemon = do actM oalg (Right Stdout); actM oalg (Left Stdout); daemon
    in do resM oalg Stdout (parM oalg p daemon))
  (interpretAT1 $ λ (Tell s k) → do act (Left Stdout); tell s; act (Right Stdout); return k)
```

The reader may roughly recognise that the code Runner $ λ ⋯ is doing what we want it to do: running a daemon in parallel with the user program p, but all operations now end with an additional M and have a parameter *oalg*. This will be explained later when we see the definition of Handler. The handler lock can be used by wiring it before our handler ccsTell: the output of running ex_2 = handleIO′ ioPar (lock ▷ ccsTell) br reads like `Oh poor boyI am...`.

Now suppose that we find that it is a bit hard to recognise which part of the output is from which process. We can solve this problem by defining a handler that prepends each Tell-operation with a *process ID*. We do not have a concept of process IDs in CCS, but we can implement it using a handler of Par that uses the reader effect [Ask String, Local String] to track the current ID:

```
tellId :: Handler [Tell String] [Tell String, Ask String] [] a a
tellId = interpret1 $ λ (Tell s k) → do id ← ask; tell (id ++ ": " ++ s ++ ". "); return k

threadId :: Handler [Par] [Par, Local String] [] a a
threadId = interpretM1 $ λ alg (Par a b) →
  parM alg (localM alg (++"L") a) (localM alg (++"R") b)
```

Composing all handlers together, we have our final jumbo handler

```
((threadId ▷ tellId) \\ reader "") ▷ lock ▷ (ccsIO \\ maybe \\ reader empty) ▷ tellIO        (5)
```

Running it with br gives us `R: Oh poor boy. LL: I am just a poor boy. LR: I need no sympathy.`, or the other two interleavings: `LL...R...LR` and `LL...LR...R...`. Although this example is artificial, we hope that it still demonstrates the fun and flexibility of composing handlers.

2.5 Light Staging

Our handler (5) above is *static*, and indeed in most use cases of effect handlers, the handler is either completely static like (5) or at least *structurally static* like in

$$\mathsf{fun}\ n = \mathbf{handle}\ \cdots\ \mathbf{with}\ (\mathsf{reader}\ n \triangleright \mathsf{maybe})$$

where the handler depends on a runtime value n but its overall structure is still static. Handlers that have a truly dynamic structure, such as in

$$\mathsf{bun}\ n = \mathbf{handle}\ \cdots\ \mathbf{with}\ h\ n\ \mathbf{where}\ h\ 0 = \mathsf{maybe}; h\ n = \mathsf{reader}\ n \triangleright h\ (n-1)$$

are rare in practice. When handlers are structurally static, handler combinators such as \\ and ▷ in principle can be partially evaluated away at compile time. Unfortunately, current versions of GHC do not perform any simplification for operations on arrays or finger trees. If compiler optimisation does not do this for us automatically, we can do it ourselves:

Idea 3 (Light Staging). We use meta-programming to evaluate away handler combinators at compile time, thus eliminating the runtime cost of composing handles.

Typed Template Haskell is a type-safe way to do meta-programming in Haskell. When it is enabled, every value is at either the compile-time stage (stage 0) or the run-time stage (stage 1). There is a type CodeQ a for *code of expressions of type* a, where a is any Haskell type. An element of CodeQ a at stage 0 is created by quoting[2], $[\![\mathsf{exp}]\!]$, where exp is a Haskell expression of type a at stage 1. An element c :: CodeQ a at stage 0 is used by splicing, $\$c$, giving us a value of type a at stage 1. Splicing and quoting are mutual inverses. Top-level bindings can be used at both stages. Intensional analysis of code is not supported in Typed Template Haskell.

To perform handler composition at the compile-time stage, we unfortunately cannot just use the type CodeQ (Handler *effs oeffs ts a b*), because elements of this type are opaque 'blobs of handlers', whose internal structure we cannot inspect or manipulate. For this reason, we have another type HandlerC *effs oeffs ts a b* (suffix C for Code), an element of which consists of the code for a runner and a (heterogeneously-typed) *compile-time* list whose entries are the handling code for an effect in *effs* (and producing *oeffs*). All our handler combinators also have a separate compile-time version that works with HandlerC instead of Handler.

As an example, if we would like to stage the handler (5), we need to replace the handler combinators ▷ and \\ with their staged counterparts ▷$ and \\$, as well as replacing the basic handlers reader and maybe with their staged counterparts of type HandlerC. Using the staged counterpart of (5) to handle our program br would then generate the following ex_3 :: IO (Maybe ()).

```
ex₃ = let alg :: Algebra [Par, Tell String, Act HS, Res HS]
                         (ReaderT String (ReaderT (QSemMap HS) (MaybeT IO)))
          alg = (λ(Par l r) → ⋯) :# (λ(Tell s k) → ⋯) :# (λ(Act a k) → ⋯) :# ⋯
      in (runMaybeT ∘ flip runReaderT empty ∘ (λ p → let daemon = ⋯ in ⋯)
            ∘ flip runReaderT "") (eval alg br)
```

The staged handler combinators statically construct an algebra for [Par, Tell String, Act HS, Res HS] on the monad ReaderT String (⋯ IO). The program br is evaluated with this algebra using eval :: Monad $m \Rightarrow$ Algebra *effs* $m \to$ Prog *effs* $a \to m\ a$. Finally the result is run with the initial values of the reader effects as well as the daemon process. In this way, the runtime overhead of handler combinators is completely eliminated.

[2]Haskell uses different concrete syntax for different kinds of quotes, but we will simply use $[\![\cdots]\!]$ for all quotes.

2.6 Full Staging

However, the program ex_3 is still not optimised to its full potential: ex_3 still involves a syntactic program br and evaluates br with the semantic operations at runtime. In contrast, if we only care about performance but not modularity, we can write a program of type IO (Maybe ()) invoking IO-primitives directly. Can we have the best of both worlds – good performance of hand-rolled code and the flexibility of effect handlers?

Idea 4 (Full Staging). We use effect handlers *at compile time* to generate code of performant low-level monadic programs that do not use effect handlers or comparable abstractions.

How to realise this idea is not straightforward though. A fundamental difficulty is that *code of a monad is not a monad*: given a monad m, we would like to construct elements of type CodeQ $(m\ a)$ at compile time, but $a \mapsto$ CodeQ $(m\ a)$ is not a monad. Although we have a 'static bind' operation

$$\begin{aligned}&\mathsf{sbind} : \mathsf{CodeQ}\ (m\ a) \to (\mathsf{CodeQ}\ a \to \mathsf{CodeQ}\ (m\ b) \to \mathsf{CodeQ}\ (m\ b)\\ &\mathsf{sbind}\ ma\ kmb = [\![\ \mathbf{do}\ a \leftarrow \$(ma); \$(kmb\ [\![a]\!])]\!]\end{aligned}$$

the type of this operation is not the type of a monadic bind because kmb takes in CodeQ a instead of a. Therefore, if we directly work with CodeQ $(m\ a)$, our meta-programs cannot be written in the monadic style, and consequently cannot integrate with algebraic effects and handlers directly (the semantic structure of computation with algebraic effects is still a monad).

A breakthrough to this problem is made by Kovács [2024]. Instead of directly working with CodeQ $(m\ a)$ at compile time, Kovács proposes working with some other monad n that comes with the following two functions relating it to m:

$$\mathsf{down} :: n\ (\mathsf{CodeQ}\ a) \to \mathsf{CodeQ}\ (m\ a), \qquad \mathsf{up} :: \mathsf{CodeQ}\ (m\ a) \to n\ (\mathsf{CodeQ}\ a).$$

Specifically, if m is the monad Identity, Kovács's choice of n is a CPS-based *code generation monad* Gen, which comes with certain operations for code generation, e.g. generating **let**-bindings. If m is StateT s Identity, then the corresponding n is StateT (CodeQ s) Gen; note here the state type is CodeQ s instead of s. The power of Kovács's approach is that it enables effectful operations to be partially evaluated at *compile time*. For example, if n is StateT (CodeQ Int) Gen, we have

$$\mathsf{down}\ (\mathbf{do}\ s \leftarrow \mathsf{get}; \mathsf{put}\ [\![\$s + 1]\!]; \mathsf{put}\ [\![\$s + \$s]\!]; \mathsf{get}) \qquad \text{generates} \qquad \boldsymbol{\lambda}\, s \to (s + s, s + s)$$

where we do not see the get and put operations in the generated code because they are evaluated at compile time. For this reason, Kovács's approach sometimes generates code even more optimised than hand-rolled programs, while retaining the monadic programming interface.

In his work [2024], Kovács works solely with monads and does not employ effect handlers. In this paper, we show that Kovács's approach can be fruitfully integrated with higher-order-effect handlers, enabling us to generate highly optimised code while keeping the modularity benefits of effect handlers. We will refer to the resulting approach fully staged Effective.

There are limitations to the fully staged approach. Firstly, not all monads m have a simple choice of the corresponding compile-time monads n. Secondly, if the programmer wants to generate high-quality code, they have to explicitly work with a few tricky (but interesting!) operations for controlling code generation, such as the ones for generating tail-calls and join points. Therefore the trade-off between the fully staged approach and the lightly staged/non-staged approach is the ease of programming and the performance.

Coming back to our running example br, if we refactor the handler ccsTell following the fully staged approach, then our earlier example ex_1 becomes

$$\mathsf{ex}_4 = \$(\mathsf{stageHML}\ \_\ \_\ ((\mathsf{ccsGenIO} \setminus\!\setminus \mathsf{reader}\ \mathsf{empty} \setminus\!\setminus \mathsf{maybe}) \triangleright \mathsf{writerGenIO})\ \mathsf{br})$$

which generates the following code (slightly re-formatted) of type IO (Maybe ()):

```
do forkIO $ do putStrLn "Oh poor boy"; return ()
   x1 ← newQSem 0; x2 ← newQSem 0
   forkIO $ do signalQSem x1; waitQSem x2; putStrLn "I need no sympathy"
   putStrLn "I am just a poor boy"; waitQSem x1; signalQSem x2
   return (Just ())
```

The generated code is as optimised as one can hope (and this example is simple enough so that the program br does not have to be changed at all). Notably, we do not see any Local or Ask operations in the generated code, because they are evaluated away at compile time.

3 OPERATIONS, ALGEBRAS, PROGRAMS

After the extended overview of Effective in Section 2, in the rest of this paper we will discuss Ideas 1 to 4 in greater details. In this section, we start with showing how we represent effectful operations and programs, which will be the foundation for everything else in the rest of the paper.

3.1 Effect Signatures

Following the previous work on higher-order effects [van den Berg and Schrijvers 2024; Wu et al. 2014; Yang and Wu 2023, 2026], an effectful operation is semantically a (pure) function of type $o :: \forall a.\ (h\ m)\ a \to m\ a$ for some monad m and 'higher-order functor' $h :: (\mathsf{Type} \to \mathsf{Type}) \to (\mathsf{Type} \to \mathsf{Type})$, which is an endofunctor on Haskell Functors, codified as the following typeclass:

```
class (∀f. Functor f ⇒ Functor (h f)) ⇒ HFunctor h where
  hmap :: (Functor f, Functor g) ⇒ (∀x. f x → g x) → (∀x. h f x → h g x)
```

We call h the *(higher-order) signature* of this effectful operation o. We abbreviate

```
type Effect = (Type → Type) → (Type → Type)
```

Example 3.1. An effectful operation taking an Int as input and producing a String as output has the signature $h\ m\ a = (\mathsf{Int}, \mathsf{String} \to a)$ where the argument m is ignored; the HFunctor instance of this h is the obvious one. The type of operations $\forall a.\ h\ m\ a \to m\ a$ for this signature functor h is then $\forall a.\ (\mathsf{Int}, \mathsf{String} \to a) \to m\ a$, which is isomorphic to $\mathsf{Int} \to m\ \mathsf{String}$ by Curry and Yoneda (assuming a parametric interpretation of $\forall$-quantification). Operations of type $\mathsf{A} \to m\ \mathsf{B}$ for some types A and B are known as *generic operations* [Plotkin and Power 2003].

Example 3.2. Binary effectful operations $b :: \forall a.\ m\ a \to m\ a \to m\ a$ that commute with sequential composition, i.e. $(b\ x\ y \gg\!= f) = b\ (x \gg\!= f)\ (y \gg\!= f)$, are equivalent to operations of type $\forall a.\ (a, a) \to m\ a$ without this equation [Jaskelioff and Moggi 2010, Remark 3.3]. Therefore they can be given the higher-order signature $h\ m\ a = (a, a)$, where the argument m is ignored. Slightly more generally, an n-ary operation $\forall a.\ m\ a \to \cdots \to m\ a$ that commutes with sequential composition can be given the higher-order signature $h\ m\ a = (a, \ldots, a)$ with the obvious instance of HFunctor.

What is common in the two examples above is that the signature h maps any m constantly to some functor $s :: \mathsf{Type} \to \mathsf{Type}$, in these two examples $s\ a = (\mathsf{A}, \mathsf{B} \to a)$ and $s\ a = (a, \ldots, a)$ respectively. An operation of type $\forall a.\ s\ a \to m\ a$ is called an *algebraic operation* by Jaskelioff and Moggi [2010], and the functor s may be called the *first-order* signature functor of this operation, to differentiate from the corresponding *higher-order* signature functor $h\ m\ x = s\ x$.

Algebraic operations are extremely common in practice and they enjoy some special properties because of their particularly simple types. Therefore Effective provides a generic higher-order signature Alg s for algebraic operations of first-order signature s:

```
newtype Alg (s :: Type → Type) (m :: Type → Type) x = Alg (s x)
```

The benefit of having Alg s is that we can define general constructions for Alg s, saving us the need to repeat the work for each algebraic operation.

Example 3.3. An operation of type $\forall a.\ m\ a \to m\ a \to m\ a$ on some monad m has the higher-order signature $h\ m\ a = (m\ a, m\ a)$. More generally, an operation $\forall a.\ m\ a \to \cdots \to m\ a$ that takes n-computations as input has signature $h\ m\ a = (m\ a, \cdots, m\ a)$. This h can also be written as $h\ m = \mathsf{I}_n \circ m$ where $\mathsf{I}_n\ x = (x, \cdots, x)$ is the functor mapping every x to the n-fold product of x.

Yet more generally, we can consider higher-order signatures $h\ m = s \circ m$ for any functor s. Operations of such signatures are called *first-order operations* by Jaskelioff and Moggi [2010] and *scoped operations* by Wu et al. [2014]. The terminology 'scoped operations' comes from the intuition that an operation call $o\ c_1\ \cdots\ c_n$ to $o :: m\ a \to \cdots \to m\ a$ is an effect acting on the computations $c_1, \cdots, c_n$. Following the common concrete syntax of using $\{\ldots\}$ to delimit scopes, we usually write $o\ \{c_1\}\ \{c_2\}\ \ldots\ \{c_n\}$ for scoped operations. The difference between scoped operations and algebraic operations (Example 3.2) is that scoped ones do not need to commute with sequential compositions. For example, parallel composition $(||) :: m\ a \to m\ a \to m\ a$ in concurrent programming is scoped but not algebraic because

$$\mathbf{do}\ x \leftarrow (\{p\}\ ||\ \{q\}); k \quad \neq \quad \{\mathbf{do}\ x \leftarrow p; k\}\ ||\ \{\mathbf{do}\ x \leftarrow q; k\}.$$

Scoped operations are common in practice too and are reasonably well-behaved (although not as nice as algebraic ones). Effective has a generic higher-order signature Scp s for them:

```
newtype Scp (s :: Type → Type) (m :: Type → Type) k = Scp (s (m k))
```

Example 3.4. Algebraic and scoped operations are the two simplest and most common families of effectful operations, but there are many operations that do not fall under these two families. For example, we may want an alternative formulation of parallel compostion $(||') :: \forall a.\ m\ a \to m\ a \to m\ (a, a)$ where the results of the two sub-computations are kept as a pair, or even $(||'') :: \forall a\ b.\ m\ a \to m\ b \to m\ (a, b)$ where the sub-computations return different types. These can also be implemented in Effective using suitable higher-order signatures. The formulation $||'$ is an instance of *parallel effects* [van den Berg and Schrijvers 2024; Xie et al. 2021], and the formulation $||''$ is studied under *concurrent monads* by Rivas and Jaskelioff [2019]; Rivas and Uustalu [2024].

Template Haskell Support. With Alg s and Scp s, when we need a new algebraic or scoped operation, we just need to define the first-order signature $s ::$ Type $\to$ Type. But even this can be considerable burden for users, so Effective provides Template Haskell helpers makeGen, makeAlg for declaring generic/algebraic operations and makeScp for declaring scoped operations (which we saw in Section 2.4). These helpers not only generate first-order signatures but also some useful type synonyms and effectful operations on the Prog monad that we will see very shortly.

Effect Sets. When more than one effect is in play, we use (type-level) lists [Effect] of effect signatures to track them. Conceptually, these lists should be understood as *finite sets* of effects and the interface of Effective works with them using set-operations (such as union and intersection) rather than list-operations. For example, the fusion of a handler h_1 consuming effects $effs_1 ::$ [Effect] and a handler h_2 consuming effects $effs_2 ::$ [Effect] is a handler consuming effects $effs_1 \cup effs_2$.

Interpreting [Effect] as sets means that an effect can only occur in an effect set at most once, but sometimes we may need more than one instance of an effect in a program. For example, a program may need two different mutable states. The traditional way of achieving this in Haskell is to let the user define **newtype** wrappers of an existing effect to copy it, but this is clearly suboptimal and we solve this by tagging existing effects with names:

```
newtype (@) (n :: Symbol) (eff :: Effect) (f :: Type → Type) (x :: Type) = WN (eff f x)
```

where a Symbol in Haskell is a type-level string. The effect n @ eff is only *nominally* different from eff, and Effective provides functions for casting handlers of eff to handlers of n @ eff. Our Template Haskell helpers also generate functions for invoking named operations. For example, $(makeScp [|par :: 2|]) generates not only par for invoking Par but also an operation parN such that parN "foo" p q invokes the effect "foo" @ Par.

3.2 An Efficient Representation of Effectful Programs and Algebras

Now we move on to defining the type Prog *effs* a of programs invoking effects in *effs* :: [Effect] and returning values of type a. A naive definition would be generalising the *data types á la carte* approach (free monads of coproducts of functors) [Swierstra 2008] to the higher-order setting:

```
data Prog_naive (effs :: [Effect]) (a :: Type) where
  Ret  :: a → Prog_naive effs a
  Call :: Sum effs (Prog_naive effs) b → (b → Prog_naive effs a) → Prog_naive effs a
```

where Sum *effs* is the coproduct of the members of *effs*:

```
data Sum (effs :: [Effect]) (m :: Type → Type) (a :: Type) where
  Yes :: eff m a        → Sum (eff : effs) m a
  No  :: Sum effs m a → Sum (eff : effs) m a
```

An implementation of effects *effs* on a monad m, called an *effs-algebra* on m, then have type

```
type Algebra_naive effs m = ∀a. Sum effs m a → m a
```

However, this representation of programs has bad time complexity for two reasons:

(1) The coproduct Sum *effs* is slow: injecting a member of *effs* into Sum *effs* introduces constructors No (No (No ⋯ Yes)) of size linear to the length of *effs*. Consequently algebras are slow too, as they need to repetitively pattern match the No constructors.
(2) Sequential composition of Prog$_{naive}$ is slow: computing $p \gg\!= k$ needs to traverse the whole structure of p to find the leaf nodes Ret a and replace them with k a, similarly to how list concatenation $xs +\!\!+ ys$ works by recursion on xs.

Algebras as Sequences. To solve these two problems, a key observation is that an element of Algebra$_{naive}$ [$eff_1, eff_2, \cdots, eff_n$] m is the same as a sequence of functions

$$(\forall a.\ eff_1\ m\ a \to m\ a,\ \ \forall a.\ eff_2\ m\ a \to m\ a,\ \ \cdots,\ \ \forall a.\ eff_n\ m\ a \to m\ a) \tag{6}$$

by the universal properties of Sum and ∀. Therefore a better representation of Algebra$_{naive}$ *effs* m is storing this sequence of functions using some efficient data structure, such as arrays or finger trees. In Effective we do not commit to any fixed data structure but define a typeclass

```
class Functor s ⇒ Sequence (s :: Type → Type) where ...
```

with common sequence operations such as nil :: $s\ a$, cons :: $a \to s\ a \to s\ a$, index :: $s\ a \to$ Int $\to a$, and concat :: $s\ a \to s\ a \to s\ a$. Then our type of algebras becomes

```
newtype Algebra_ (s :: Type → Type) effs m = Algebra (s Any)
```

where Any is a type in Haskell for bypassing type checking. Unfortunately we need it because almost all the existing efficient data structures in Haskell store elements of the same type, but the elements of our sequence (6) have different types. However, we do not export the internal representation of Algebra_ but only a safe interface, so we are still type-safe. We use the finger tree implementation from the `containers` package [Haskell Contributors 2025] as a default choice of s:

```
type Algebra effs f = Algebra_ Data.Sequence.Seq effs f
```

which gives us amortised $O(1)$ consing and worst-case log-time concatenation and accessing.

The safe interface of Algebra that we expose includes functions for building algebras, such as

```
endAlg :: Sequence s ⇒ Algebra_ s [ ] m
(:#) :: Sequence s ⇒ (∀x. eff m x → m x) → Algebra_ s effs m → Algebra_ s (eff : effs) m
```

and functions for accessing the entries of an algebra, such as

```
dispatch :: (eff ∈ effs, Sequence s) ⇒ Algebra_ s effs m → ∀x. eff m x → m x
```

Impredicative Encodings of Programs. There are a few known ways of making $\text{Prog}_{\text{naive}}$ efficient, such as using codensity transformation [Voigtländer 2008] and the technique of Ploeg and Kiselyov [2014]. Because Effective is designed to be a library of *deep handlers*, we only need to support sequential composition of programs efficiently and do not need to worry about the cost of pattern matching of programs (which is needed for shallow handlers). Therefore a very simple impredicative encoding of programs is sufficient:

```
newtype Prog (effs :: [Effect]) (a :: Type) = Prog
  {runProg :: ∀m s. (Monad m, Sequence s) ⇒ Algebra_ s effs m → m a}
```

The type constructor Prog *effs* can be given a monadic interface (return and ≫=) by using that of m and it supports operations from *effs* :: [Effects] as follows:

```
call :: (eff ∈ effs, HFunctor eff) ⇒ eff (Prog effs) a → Prog effs a
call x = Prog (λ alg → dispatch alg (hmap (λ p → runProg p alg) x))
```

The recursion principle of effectful programs is that whenever we have a monad m equipped with an *effs*-algebra, we can evaluate a program with effects *effs* into m. For our impredicative-encoded Prog *effs a*, this is almost trivial:

```
eval :: (Sequence s, Monad m) ⇒ Algebra_ s effs m → Prog effs a → m a
eval alg p = runProg p (toArrayAlgebra alg)
```

For performance we convert the algebra *alg* to an array-based version using toArrayAlgebra :: Sequence s ⇒ Algebra_ s *effs* m → Algebra SmallArray *effs* m before supplying it to the program, because programs built from call and the monad interface only read the entries of an algebra and never construct algebras, so switching to arrays as the sequence structure is optimal.

Example 3.5. A standard example of computational effects is *mutable state*. We declare the operations for writing and reading a state by

```
$(makeGen [|put :: ∀s. s → ()|])        $(makeGen [|get :: ∀s. s|])
```

which generate the following package of things for working with these two operations:

(1) signatures Put, Get :: Type → Effect parameterised by the type of state; in fact, the signature Put s is a synonym of Alg (Put_ s) where **data** Put_ s x = Put_ s x, so any constructions for algebraic operations Alg can be applied to Put s as well, similarly for Get s;
(2) constructors Put :: s → x → Put s m x and Get :: (s → x) → Get s m x;
(3) operations put :: Put s ∈ *effs* ⇒ s → Prog *effs* () and get :: Get s ∈ *effs* ⇒ Prog *effs* s as well as their named variants putN and getN;
(4) operations putM :: Put s ∈ *effs* ⇒ Algebra *effs* m → s → m () that works on any m that comes with an algebra of Put s and similarly getM for Get s.

As an example of effectful programs, the following is Euclid's algorithm using two mutable integers:

```
euclid :: ([Put Int, Get Int, "n" @ Put Int, "n" @ Get Int] ⊆ effs) ⇒ Prog effs Int
euclid = do a ← get; b ← getN "n"
            if b ≡ 0 then return a else do put b; putN "n" (a `mod` b); euclid
```

For the sake of demonstration, we have made only one state named, so get and put use effects [Put Int, Get Int] while getN "n" and putN "n" use effects ["n" @ Put Int, "n" @ Get Int].

By representing algebras as sequences and the impredicative encoding of programs, we have eliminated the inefficiency (1) and (2) of $\text{Prog}_{\text{naive}}$. Moreover, when optimisation is enabled, GHC can usually specialise and inline the parameters *m* and *s*, eliminating the cost of Prog altogether. However, there is no guarantee that GHC always does this, which is why we use meta-programming to optimise the performance ourselves in Section 5.

4 ALGEBRA TRANSFORMERS AND HANDLERS

Prog and Algebra are already a minimalist effectful programming interface: operations are declared as HFunctors (or using the Template Haskell helpers); effectful programs are constructed using call and the monad interface; elements of Algebra *effs m* are constructed using functions such as (:#) and are applied to programs using eval; in particular the carrier monad *m* may be the monad Prog *effs*′ for some *effs*′ :: [Effect] or other existing monads in Haskell such as IO.

What this interface still lacks is *semantic modularity*. For example, if we want to define algebras

```
Algebra [Put s, Get s] (StateT s Identity)    Algebra [Put s, Get s, Throw] (StateT s Maybe)
```

on the monads StateT Identity and StateT Maybe where

```
newtype StateT s m a = StateT {runStateT :: s → m (a, s)}
```

for now we have to repeat the definitions of the operations Get *s* and Put *s* on these two monads.

To define algebras modularly, we propose *algebra transformers* (Section 4.1), which are inspired by Moggi's [1989b] *monad transformers* and Yang's [2024] *model transformers*. The combinators of algebra transformers in Sections 4.2 to 4.4 are the core novelty of Effective. In Section 4.5, a concept of *handlers* is defined, which are algebra transformers paired with *runners*.

4.1 Definition and Some Examples

Algebra transformers are related to Moggi's [1989b] *monad transformers*: a monad transformer *t* is a function mapping every monad *m* to a monad *t m* (together with a monad morphism $m \to t\ m$ called lift in Haskell), and an algebra transformer is a function from algebras to algebras:

```
newtype AlgTrans effs oeffs ts cs = AlgTrans
  {getAT :: ∀m. cs m ⇒ Algebra oeffs m → Algebra effs (Apply ts m)}
```

where *effs*, *oeffs* :: [Effect], *ts* :: [(Type → Type) → (Type → Type)], and *cs* :: (Type → Type) → Constraint is a typeclass constraint on Type → Type. The type-level function Apply :: $[k \to k] \to k \to k$ applies a list of (type-level) functions to an element:

$$\text{Apply}\ [\,]\ x = x, \qquad \text{Apply}\ (f : fs)\ x = f\ (\text{Apply}\ fs\ x).$$

Typically, the index *ts* will be a list of monad transformers and *cs* will be just the constraint Monad. In this case, an algebra transformer AlgTrans *effs oeffs ts* Monad is a function mapping *oeffs*-algebras on *m* to an *effs*-algebra on the monad Apply *ts m*, for every monad *m* polymorphically.

Evaluating a program with an algebra transformer is done by the following function:

```
evalAT :: (cs m, oeffs ⊆ xeffs, Monad (Apply ts m))
       ⇒ Algebra xeffs m → AlgTrans effs oeffs ts cs → Prog effs a → Apply ts m a
evalAT oalg at = eval (getAT at (weakenAlg oalg))
```

where the typeclass $oeffs \subseteq xeffs$ provides `weakenAlg :: Algebra xeffs m → Algebra oeffs m`. Similarly to how $eff \in effs$ works, $oeffs \subseteq xeffs$ is resolved inductively.

Example 4.1. An algebra transformer over the state monad transformer `StateT s` is as follows:

```
stateAT :: AlgTrans [Put s, Get s] [] [StateT s] Monad
stateAT = AlgTrans $ λ _ → (λ (Put s k) → StateT (λ _ → return (k, s))) :#
                           (λ (Get k) → StateT (λ s → return (k s, s))) :# endAlg
```

Example 4.2. In the last example `stateAT` transforms a trivial algebra `Algebra [] m` into an algebra `Algebra [Put s, Get s] (StateT s m)`. The following is another possibility:

```
log :: Show s ⇒ AlgTrans [Put s, Get s] [Put s, Get s, Tell String] [] Monad
log = AlgTrans $ λ oalg → (λ (Put s k) → do tellM oalg ("put " ++ show s); putM oalg s)
  :# (λ (Get k) → do s ← getM oalg; tellM oalg ("got " ++ show s); return s) :# endAlg
```

This one transforms an existing `[Put s, Get s, Tell String]`-algebra on a monad m into a new algebra on the same monad m by inserting a call to `Tell` for each call to `Put` and `Get`.

If we view `log` as an effect handler, `[Put s, Get s]` is the effects that it handles, and the effect set `[Put s, Get s, Tell String]` is the effects that this handler produces. Therefore, we call the parameter $effs$ of a transformer `AlgTrans effs oeffs ts cs` the *input effects* and $oeffs$ the *output effects*.

Example 4.3. Suppose we want to handle some effect eff with output effects $oeffs$. One way to describe the handler is to simply write an effectful program with effects $oeffs$. This is done by the following combinator, which appeared in the `lock` example in Section 2.4:

```
interpretAT1 :: HFunctor eff ⇒ (∀m x. eff m x → Prog oeffs x)
             → AlgTrans [eff] oeffs [] Monad
interpretAT1 f = AlgTrans $ λ oalg → (eval oalg ∘ f) :# endAlg
```

There is also a function `interpretAT` in Effective for re-interpreting an effect set $effs$ rather a single effect eff. The transformer `log` in Example 4.2 can be alternatively defined with `interpretAT`.

4.2 Structural Combinators of Algebra Transformers

Next we have a few combinators that essentially say `AlgTrans` forms a *graded promonad* over the poset of effect sets, and we call them *structural combinators*. In this paper we will only follow the categorical structure as an informal guidance. A more formal account is left as future work.

We have identity transformers and composition of transformers:

```
idAT :: AlgTrans effs effs [] cs
idAT = AlgTrans $ λ alg → alg

compAT :: (∀m. Assoc ts1 ts2 m) ⇒ AlgTrans effs1 effs2 ts1 cs1 → AlgTrans effs2 effs3 ts2 cs2
       → AlgTrans effs1 effs3 (ts1 ++ ts2) (CompC ts2 cs1 cs2)
compAT at1 at2 = AlgTrans $ λ (oalg :: Algebra effs3 m) → getAT at1 (getAT at2 oalg)
```

The constraint `CompC ts2 cs1 cs2 m` of the resulting transformer is defined to be the conjunction of `cs2 m` and `cs1 (ts2 m)`, because m is supplied to alg_2 while `ts2 m` is supplied to alg_1.

The constraint `Assoc ts1 ts2 m` of `compAT` is defined to be `Apply ts1 (Apply ts2 m)~Apply (ts1 ++ ts2) m`, which is needed for `compAT` to type check. This constraint is automatically satisfied for any

concrete values of ts_1 and ts_2, but unfortunately the type system of Haskell is not powerful enough for us to prove this fact internally and remove the constraint from the type of compAT. If we had implemented Effective as a standalone language or in a dependently typed language, there would not be superficial constraints such as $\forall m.\ \mathsf{Assoc}\ ts_1\ ts_2\ m$. For clarity we will elide such superficial constraints as dots $\cdots \Rightarrow$ in the rest of the paper.

The next combinator is weakening: given some AlgTrans *effs oeffs ts cs*, we can also use it to handle less effects $effs' \subseteq effs$ while being allowed to produce more effects $oeffs' \supseteq oeffs$ and receiving carriers satisfying a stronger constraint cs' that implies *cs*:

```
weakenAT :: (effs' ⊆ effs, oeffs ⊆ oeffs', ∀m. cs' m ⇒ cs m)
         ⇒ AlgTrans effs oeffs ts cs → AlgTrans effs' oeffs' ts cs'
weakenAT at = AlgTrans $ λ oalg → weakenAlg (getAT at (weakenAlg oalg))
```

Next we have a combinator that performs case splits on the input effects:

```
caseAT :: ··· ⇒ AlgTrans effs₁ oeffs ts cs₁ → AlgTrans effs₂ oeffs ts cs₂
       → AlgTrans (effs₁ ∪ effs₂) oeffs ts (cs₁ ∧ cs₂)
caseAT at₁ at₂ = AlgTrans $ λ oalg → unionAlg (getAT at₁ oalg) (getAT at₂ oalg)
```

These a few combinators above outline the fundamental structure of algebra transformers. For a smoother user interface, Effective also provides variants of those fundamental combinators, which we will not discuss in detail here.

4.3 Canonical Forwarding of Effects

One common situation of algebra transformers is when we have some effects *effs* :: [Effect] on some carrier *m* :: Type → Type and we have some *t* :: (Type → Type) → (Type → Type), we would like to 'forward' or 'lift' the effects *effs* on *m* to effects on *t m*. For some specific *effs* and *t*, *but not in general*, there is a canonical way to forward effects. Therefore Effective has a typeclass Forwards *effs ts* when a canonical forwarder for *effs* and *ts* exists:

```
class Forwards (effs :: [Effect]) (ts :: [(Type → Type) → (Type → Type)]) where
  type ForwardsConstraint effs ts :: (Type → Type) → Constraint
  fwds :: AlgTrans effs effs ts (ForwardsConstraint effs ts)
```

The typeclass Forwards *effs ts* is inductively resolved by forwarding each element of *effs* along each element of *ts*, reducing Forwards *effs ts* to the following typeclass Forward *eff t* that forwards one effect along one transformer. This Forward typeclass is where we *define forwarders*, while Forwards is a nicer interface for *using forwarders*.

```
class Forward (eff :: Effect) (t :: (Type → Type) → (Type → Type)) where
  type FwdConstraint eff t :: (Type → Type) → Constraint
  fwd :: FwdConstraint eff t m ⇒ (∀x. eff m x → m x) → (∀x. eff (t m) x → t m x)
```

When the effect is an algebraic operation and *t* is a monad transformer, we have a canonical forwarder as follows. The fact that algebraic operations can always be canonically forwarded is the semantic reason why handlers of algebraic operations exhibit exceptional modularity.

```
instance MonadTrans t ⇒ Forward (Alg f) t where
  type FwdConstraint (Alg f) t = Monad
  fwd alg (Alg o) = lift (alg (Alg o))
```

However, forwarding non-algebraic operations is trickier and generally does not have a canonical choice. Jaskelioff and Moggi [2010] did a systematic study of forwarding scoped operations (called

first-order operations op. cit.) and they show that scoped operations can be forwarded along *functorial transformers*, which include StateT, ReaderT, WriterT, ExceptT, ListT, and the resumption monad transformer (also known as the 'free monad transformer') ResT $s\ m\ a = m\ (a + s\ (\mathsf{ResT}\ m\ a))$ as examples. Effective implements this forwarder for StateT, ReaderT, WriterT and ExceptT but for ListT and ResT, Jaskelioff and Moggi [2010]'s forwarder usually is *not* the desirable one in our programming examples. Therefore it is not declared as a Forward-instance for ListT and ResT, and we declare a more conservative forwarder of only *unary* scoped operations for ListT and ResT.

Our main way of using the canonical forwarders is through the following combinator, which makes effects *feffs* bypass an Algebra *effs oeffs ts cs* provided that *feffs* can be forwarded along *ts*.

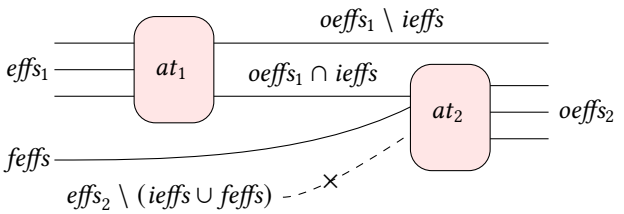
```
type ForwardsC cs feffs ts = (Forwards effs ts, ∀m. cs m ⇒ ForwardsConstraint effs ts m)
withFwds :: (··· , ForwardsC cs feffs ts) ⇒
  AlgTrans effs oeffs ts cs → AlgTrans (effs ∪ feffs) (oeffs ∪ feffs) ts cs
withFwds at = weakenAT (unionAT at (fwds @feffs))
```

4.4 Fusion Combinators of Algebra Transformers

As we motivated in Idea 1 and Idea 2, the expressiveness of Effective mainly comes from *fusion combinators*. We saw two fusion-based combinators ▷ and \\ in the introduction, and both of them are special cases of the following *general fusion combinator*:

```
generalFuseAT :: (··· , feffs ⊆ effs2, ieffs ⊆ effs2,
      ForwardsC cs1 feffs ts1, ForwardsC cs2 (oeffs1 \ ieffs) ts2)
  ⇒ AlgTrans effs1 oeffs1 ts1 cs1 → AlgTrans effs2 oeffs2 ts2 cs2
  → AlgTrans (effs1 ∪ feffs) ((oeffs1 \ ieffs) ∪ oeffs2) (ts1 ++ ts2) (CompC ts2 cs1 cs2)
```

This combinator can be defined by composing structural combinators and withFwds (although in our actual implementation it is defined directly for better performance). The best way to show the definition of generalFuseAT is via the diagram on the right. The input to generalFuseAT are two algebra transformers at_1 and at_2 and two subsets *feffs*, *ieffs* of the input effects $effs_2$ of at_2. The argument *feffs* names the effects of at_2 that we want to expose to the upstream, while *ieffs* names the effects of at_2 that we use to intercept the effects output by at_1. If a member of $effs_2$ is neither in *ieffs* or *feffs*, it is then discarded.



The general fusion combinator has a few useful specialisations:

- If we choose both *ieffs* and *feffs* to be $effs_2$, we use the capability of at_2 in a maximal way, and this gives us the fusion combinator ▷ in the introduction, except that ▷ in the introduction works on not AlgTrans but Handler, which we will introduce shortly below.
- If we choose *feffs* to be the empty set while *ieffs* to be $effs_2$, we use at_2 only to handle the effects produced by at_1 but not the effects from the upstream. This is essentially the \\ combinator from the introduction, except that \\ works on Handler rather than AlgTrans.
- If we choose *ieffs* to be the empty set while *feffs* to be $effs_2$, we use at_1 and at_2 in a non-interfering way – at_2 handles effects from the upstream but not the effects produced by at_1. (If an upstream effect is both in $effs_1$ and $effs_2$, it is handled by at_1.)

As we show in the concurrency example in Section 2.4, these fusion combinators provide an expressive way to build handlers from smaller pieces.

Remark 4.4. The definition of AlgTrans and the combinators did not mention Monad at all, so our infrastructure of AlgTrans can in fact be used for other notions of computation, such as applicative functors. Our type Prog of effectful programs is specific to monads, but it is easier to replace the Monad in the definition of Prog with Applicatives or other constraints.

The carrier m in the definition of AlgTrans has kind Type → Type, so AlgTrans cannot be directly used for notions of computation that do not have this kind, such as *graded monads* [Katsumata 2014] or *arrows* [Hughes 2000]. However, we can use the trick of *phantom types* to work around this. For example, if we want to encode arrows as Type → Type, we can define a trivial datatype **data** Prod $(a ::$ Type$)$ $(b ::$ Type$)$ that encodes a pair of types as one type and also type families Fst, Snd :: Type → Type such that Fst (Prod a b) = a and Snd (Prod a b) = b. Then an arrow $p ::$ Type → Type → Type can be encoded as $p' ::$ Type → Type with p' $a = p$ (Fst a) (Snd a).

4.5 Runners and Handlers

The programming interface that we have seen so far is as follows: (1) the programmer writes an effectful program $p ::$ Prog *effs* a and (2) builds an algebra transformer $at ::$ AlgTrans *effs oeffs ts* Monad using the combinators above (Effective also comes with a bag of basic algebra transformers for some standard effects), and finally (3) programs are evaluated using evalAT *alg at p*, where *alg* :: Algebra *oeffs m*. Typically m is either the Identity monad with *oeffs* being empty or the IO monads with *oeffs* being some primitive effects that IO supports.

The result of evalAT has type Apply *ts m a*, and we usually need to 'run' the list of carrier transformers *ts* to obtain a result that we really want. For example, *ts* may be [StateT Int, MaybeT], and we may want to turn Apply [StateT Int, MaybeT] Identity a into Maybe $(a,$ Int$)$ using the function runIdentity ∘ runMaybeT ∘ flip runStateT 42. Although final-processing steps like this typically are not very complex, it is still a win if they can be managed in a modular way.

To this end, we define the following type of *runners* that runs the carrier transformers *ts*, possibly producing effects *oeffs* :: [Effect], and changes the return-value type from a to b:

```
newtype Runner oeffs ts a b cs =
  Runner {getR :: ∀m. cs m ⇒ Algebra oeffs m → Apply ts m a → m b}
```

Our type of *handlers* processing effects *effs*, producing effects *oeffs*, and changing the return-value from type a to b is simply a pair of a runner and an algebra transformer:

```
data Handler effs oeffs ts a b =
  Handler {hrun :: Runner oeffs ts a b Monad,  halg :: AlgTrans effs oeffs ts Monad}
```

We have also specialised the typeclass constraints to Monad for a simpler user interface. Applying a handler Handler *effs oeffs ts a b* to an effectful program Prog *effs a* results in a value of type m b:

```
handleM :: (Monad m, Monad (Apply ts m), oeffs ⊆ xeffs)
  ⇒ Algebra xeffs m → Handler effs oeffs ts a b → Prog effs a → m b
handleM xalg (Handler r a) = getR r (weakenAlg xalg) ∘ evalAT xalg a
```

The function handleIO′ in Section 2.4 is variant of handleM that specialises m to be IO and forwards the effects *xeffs* along the handler so that it can be used by the input effectful programs as well.

The type AlgTrans is the central construct of Effective, while Handler is a slightly more specialised concept. They have a set of combinators very similar to those of AlgTrans, which are defined using the combinators of AlgTrans while composing the runners suitably. We shall not re-iterate all of the combinators, but here is the fusion combinator ▷ that appeared in Idea 1:

```
(▷) :: (∀m. Monad m ⇒ (Monad (Apply ts₁ m), Monad (Apply ts m)),
        ForwardsC Monad effs₂ ts₁, ForwardsC Monad (oeffs₁ \ effs₂) ts₂, ⋯)
    ⇒ Handler effs₁ oeffs₁ ts₁ a b → Handler effs₂ oeffs₂ ts₂ b c
    → Handler (effs₁ ∪ effs₂) ((oeffs₁ \ effs₂) ∪ oeffs₂) (ts₁ ++ ts₂) a c
```

Example 4.5. The Runner part of a handler can be used for initialisation and finalisation work. For example, with the algebra transformer stateAT from Example 4.1, we can define

```
stateIO :: IO s → Handler [Put s, Get s] [Alg IO] [StateT s] a (a, s)
stateIO ms = Handler (Runner $ λ oalg t → dispatch oalg (Alg ms) >>= runStateT t)
                     (weakenAT stateAT)
```

which obtains the initial state by an IO-action *ms*, which must be handled downstream.

5 STAGING HANDLERS AND EFFECTFUL PROGRAMMING

Because of our fusion-based combinators for handlers, and the fact that we store Algebra with efficient data structures, Effective has good time complexity for effectful programming. However, good time complexity does not automatically imply good performance in practice. In this section, we show how we can push the performance of Effective further by meta-programming. In Section 5.1, we start with a simple approach that evaluates our handler combinators at compile time. In Section 5.3, we show a more sophisticated approach that evaluates both handlers and effectful programs at compile time, while retaining a monadic interface.

5.1 Lightly Staged Effect Handlers

As we motivated in Section 2.5, in practice the handlers applied to an effectful program can usually be statically known. Thus we would hope that GHC could inline our handler combinators at compile time, simplifying handler expressions like (5) to an array of statically known entries. Unfortunately, current versions of GHC perform almost no inlining for either finger trees (because their operations are recursively defined) or arrays (because their operations are primitive).

If the compiler does not do the optimisation for us, we can always roll up our sleeves and do it ourselves by *meta-programming*. We define the following type AlgebraC for *static algebras*:

```
data AlgebraC (effs :: [Effect]) (f :: Type → Type) where
  EndAC :: AlgebraC [] f
  (:#$)  :: CodeQ (∀x. eff m x → m x) → AlgebraC effs m → AlgebraC (eff : effs) m
```

where CodeQ *a* is the type of code of an expression of type *a* in Haskell. Here we simply use heterogeneous lists, instead of finger trees or arrays, to store static algebras. This choice does not affect *runtime* performance since AlgebraC is only used at compile time.

The purpose of static algebras AlgebraC *effs m* is to generate code of algebras:

```
genAlgebra :: AlgebraC effs f → CodeQ (Algebra effs f)
genAlgebra EndAC     = ⟦endAlg⟧
genAlgebra (a :#$ as) = ⟦$a :# $(genAlgebra as)⟧
```

To build static algebras, we would like a version of AlgTrans, Handler, and their combinators that work with AlgebraC instead of Algebra. Unfortunately, in current versions of GHC there is no good way to do 'stage-polymorphic meta-programming', so we have to duplicate all our definitions to work with AlgebraC and have appropriate quoting and splicing in Effective. We will not show everything again but only the static version of AlgTrans, where Algebra is replaced by AlgebraC:

```
newtype AlgTransC effs oeffs ts cs =
  AlgTransC {getATC :: ∀m. cs m ⇒ AlgebraC oeffs m → AlgebraC effs (Apply ts m)}
```

For a concrete example, the static version of stateAT from Example 4.1 is then

```
stateATC :: AlgTransC [Put s, Get s] [] [StateT s] Monad
stateATC = AlgTransC $ λ _ → ⟦λ (Put s k) → StateT (λ _ → return (k, s))⟧ :#$
                             ⟦λ (Get k) → StateT (λ s → return (k s, s))⟧ :#$ EndAC
```

In practice, handlers most of the time have a static structure, so the light staging interface of Effective can usually be a drop-in replacement for the non-staged interface. Only on rare occasion, handlers have a truly dynamic structure, for example, when a handler is constructed according to the result of reading a configuration file.

5.2 Staged Monadic Programming à la Kovács

Light staging evaluates handler combinators at compile time, generating a statically known array of effectful operations, and a program Prog *effs a* would then index into this array (in constant time) when an operation is called. To push the performance further, we would like to eliminate the cost of looking up operations by inlining the operations at compile time. A natural idea is to replace Prog from Section 3 with the following that works with AlgebraC instead:

```
newtype ProgC effs a = ProgC (∀m. Monad m ⇒ AlgebraC effs m → CodeQ (m a))
```

However, the type ProgC *effs* does *not* have a monadic bind ≫= but only a static variant

$$\texttt{sbind} :: \texttt{ProgC}\ \mathit{effs}\ a \to (\texttt{CodeQ}\ a \to \texttt{ProgC}\ \mathit{effs}\ b) \to \texttt{ProgC}\ \mathit{effs}\ b,$$

which would significantly change the programming interface.

Even if we are willing to program with sbind, a big limitation of the approach of ProgC is that it does *not* evaluate statically known effectful operations at compile time. For example, the program sbind (put ⟦24⟧) (λ _ → put ⟦42⟧) would generate roughly the following code:

$$\texttt{StateT}\ (\lambda\ \_ \to \texttt{return}\ ((), 24)) \gg\!= (\lambda\ \_ \to \texttt{StateT}\ (\lambda\ \_ \to \texttt{return}\ ((), 42)))$$

while ideally we would like to generate simply StateT (λ _ → return ((), 42)), where the first operation is evaluated away at compile time. This may look difficult, but Kovács [2024] shows how this can be done using *staged monads*. In this subsection, we will briefly review Kovács's approach and later we will show it can be adapted to Effective.

The core of Kovács's proposal is that we use two related but different monads, one n for compile time and another m for run time. The two monads are related the following two functions:

```
class Improve n m where
  down :: n (CodeQ a) → CodeQ (m a)    up :: CodeQ (m a) → n (CodeQ a)
```

The function down generates code of a (run-time) m-program from a (compile-time) n-program, while the function up reflects code of a runtime program back to a compile-time program. For performance, the runtime monadic program should be free of any expensive abstraction such as effect handlers or mtl-style typeclasses – Kovács even disallows lambda closures in his runtime programs, but for compile-time programs we can live with the cost of some abstractions.

For generating pure code, Kovács uses a CPS-based *code generation monad* Gen at compile time:

```
newtype Gen a = Gen {runGen :: ∀r. (a → CodeQ r) → CodeQ r}
instance Improve Gen Identity where
  down g = ⟦Identity $(runGen g id)⟧    up i = Gen $ λ k → k ⟦runIdentity $i⟧
```

The monad Gen comes with operations for generating code, such as generating a **let**-binding

```
genLet :: CodeQ a → Gen (CodeQ a)
genLet c = Gen (λ k → ⟦ let x = $c in $(k ⟦x⟧)⟧)
```

and generating case splits of datatypes, for which a typeclass Split is defined for a uniform interface:

```
class Split a b where genSplit :: CodeQ a → Gen b
instance Split Bool Bool where
  genSplit cb = Gen $ λ k → ⟦ if $cb then $(k True) else $(k False)⟧
instance Split (a, b) (CodeQ a, CodeQ b) where
  genSplit cab = Gen $ λ k → ⟦ case $cab of (a, b) → $(k (⟦a⟧, ⟦b⟧))⟧
```

and so on for other datatypes. Note how genSplit conceptually turns a runtime Boolean/pair back to a compile-time Boolean/pair by generating case splits. This is sometimes called 'the trick' in meta-programming [Danvy et al. 1996].

Kovács then defines Improve-instances for common monad transformers such as MaybeT, ExcepT, ReaderT, WriterT, and StateT. Let us see StateT as an example here:

```
instance (MonadGen n, Improve n m) ⇒ Improve (StateT (CodeQ s)) (StateT s) where
  down g = ⟦StateT $ λ s → $(down (do (ca, cs) ← runStateT g ⟦s⟧; return ⟦($ca, $cs)⟧))⟧
  up cm  = StateT $ λ cs → up ⟦runStateT $cm $cs⟧ >≫= liftGen ∘ genSplit
```

where the typeclass MonadGen n has exact one member liftGen :: Gen a → n a. Given a meta-program g :: StateT (CodeQ s) n (CodeQ a), the down function generates a runtime program StateT $ λ s → ⋯ by passing the quote of the state argument ⟦s⟧ to g and recursively use the down function for n and m to generate code for the function body CodeQ (m (a, s)). On the other hand, given code of a runtime program cm :: CodeQ (StateT s m a), the up function constructs a meta-level stateful program that recursively uses up for n and m to obtain n (CodeQ (a, s)) and then generates a case split for the pair (a, s) to obtain a meta-level pair (CodeQ a, CodeQ s).

Example 5.1. The following program cd :: StateT Int Int decrements the state until it is zero:

```
cd = $(down (do s ← get; b ← liftGen (genSplit ⟦$s ≡ 0⟧)
                if b then return ⟦0⟧
                     else put ⟦42⟧; put ⟦$s − 1⟧; n ← up ⟦cd⟧; return ⟦$n + 1⟧))
```

The generated code, with Identity wrappers removed for clarity, looks like this:

```
cd = StateT $ λ s → if s ≡ 0 then return (0, s)
                             else case runStateT cd (s − 1) of (a, b) → (a + 1, b)
```

Note that we do not see the effect of put ⟦42⟧ in the generated code, because effectful operations of the meta-program are evaluated at compile time, and put ⟦$s − 1⟧ overwrites put ⟦42⟧.

5.3 Fully Staged Effect Handlers

For runtime programs, Kovács's approach generates abstraction-free monadic code, which can be viewed as 'bare metal effectful programming' in a purely functional language. However, for meta programs we do not have to tie ourselves to bare metal. Some performance cost is acceptable in exchange for expressiveness and modularity. Our plan is to use Effective to write effectful meta-programs Prog *effs* (CodeQ a) such as cd above. These are then handled using our existing AlgTrans interface, giving us compile-time monadic programs n (CodeQ a), which are finally down'ed to efficient runtime monadic programs CodeQ (m a).

Up and down effectively. The type of up :: ∀*a*. CodeQ (*m a*) → *n* (CodeQ *a*) in Improve is equivalently ∀*a x*. CodeQ (*m a*) → (CodeQ *a* → *x*) → *n x* by Yoneda, so it can be expressed as an algebraic operation **type** Up *m n x* = Alg (Up_ *m*) *n x* with the following first-order signature:

```
data Up_ m x where Up_ :: ∀a. CodeQ (m a) → (CodeQ a → x) → Up_ m x
```

The up operation for StateT from the last subsection can be generalised as the following algebra transformer, where split *as* = call (Alg (genSplit *as*)),

```
upState :: AlgTrans [Up (StateT s m)] [Up m, Put (CodeQ s), Get (CodeQ s), Alg Gen] [] Monad
upState = interpretAT1 $ λ(Alg (Up_ cm k)) →
  do cs ← get; as ← up ⟦runStateT $cm $cs⟧; (a, s) ← split as; put s; return (k a)
```

We gain some additional flexibility by only demanding Put (CodeQ *s*) and Get (CodeQ *s*) to be available output effects, rather than demanding the monad transformer to be literally StateT. For example, we may later compose upState with non-standard handlers of Put and Get. Algebra transformers of Up for ReaderT, WriterT, ExceptT, and MaybeT can be similarly defined.

On the other hand, the operation down :: *n* (CodeQ *a*) → CodeQ (*m a*) is not an operation but a *co-operation* on *n*, because *n* does not appear in the codomain at all. They cannot be treated as effectful operations in Effective, but fortunately our AlgTrans interface allows us to specify typeclass constraints on the carrier monad, so we define a new typeclass for down:

```
class n ⇝ m where down :: n (CodeQ x) → CodeQ (m x)
```

Currently in Effective instances of this typeclass are defined manually, but it should be possible to derive instances of this typeclass in a generic way, at least for non-recursive datatypes.

Staging Lists. One notable omission in Kovács [2024]'s treatment of staged monads is the *list monad transformer* for the effect of *nondeterministic choice*:

```
newtype ListT m a = ListT {runListT :: m (Maybe (a, ListT m a))}
```

Staging nondeterminism is especially useful because it amounts to generating efficient code for lists/streams of values. However, the difficulty is that ListT is a *recursive* datatype, so we cannot use 'the trick' to define up :: CodeQ (ListT *m a*) → ListT *n* (Code *a*) as we did for StateT, because we do not know how many *a*-elements there are in an element of ListT *m a* and we cannot generate infinitely many case splits. Kovács works around this problem by using a datatype Pull *a* at compile time that supports a choice operation but is *not* a monad.

Here we contribute a monad transformer PushT, which to our knowledge is novel and can be used as the compile-time monad transformer for staging lists:

```
newtype PushT n a = PushT
  {runPushT :: ∀t. (a → n (CodeQ t)) → n (CodeQ t)) → n (CodeQ t) → n (CodeQ t)}
```

The monad transformer PushT is similar to Kiselyov et al. [2005]'s CPS-based list monad transformer LogicT, except that the final answer type is restricted to code CodeQ *t* rather than an arbitrary type. The choice operations and the monad instance of PushT *n* are similar to that of LogicT. The down operation for PushT simply generates each choice one by one:

```
instance n ⇝ m ⇒ PushT n ⇝ ListT m where
  down p = ⟦ListT $(down (runPushT p
    (λca → fmap (λcas → ⟦Just ($ca, ListT (return $cas))⟧)) (return ⟦Nothing⟧))))⟧
```

What is interesting about PushT is the associated up-operation:

```
upPush :: Monad m ⇒ AlgTrans [Up (ListT m)] [Up m] [PushT] (MonadDown m)
upPush = AlgTrans $ λ oalg →
  let upMN :: ∀x. CodeQ (m x) → n (CodeQ x)
      upMN cm = dispatch oalg (Alg (Up_ cm id))
  in (λ (Alg (Up_ o ka)) → PushT $ λ kc kn → upMN
      ⟦ foldListT (λ a as → $(down (kc (ka ⟦a⟧) (upMN ⟦as⟧)))) $(down kn) $o ⟧) :# endAlg
```

where the bound variables have types o :: CodeQ (ListT m a), ka :: CodeQ $a \to x$, kc :: $x \to n$ (CodeQ t) $\to n$ (CodeQ t), and kn :: n (CodeQ t). The idea is that we generate a fold of o and use up and down operations for the monads m and n to convert between compile time and runtime as needed. The constraint MonadDown m n is the conjunction of Monad n and $n \rightsquigarrow m$.

There are a lot more that we would like to explain: programming examples of PushT, generating compact tail-recursive code, generating join points, and adapting PushT to the resumption monad transformer for concurrency. Due to the space constraint, we leave these for a future occasion.

6 DISCUSSION

Performance. The runtime overhead of Effective can be broken down into two parts: (1) the cost of invoking operations and (2) the cost of handler combinators. For (1), invoking an operation in an effectful program has constant-time overhead because algebras are converted to arrays before feeding them to programs. However, invoking an operation in a handler clause has logarithmic cost with respect to the size of effect sets, because algebras are stored in finger trees when composing handlers. For (2), the cost of a handler combinator can be further broken down into the cost of weakening algebras and the cost of appending algebras. Under the assumption that every operation will be used at least once, the cost of weakening algebras can be neglected because it can be amortised to operation lookups. The cost of appending algebras is logarithmic to the size of effect sets. Both light staging and full staging eliminate cost (2), but only full staging eliminate cost (1).

In particular, in comparison to directly writing monadic programs m :: M a with operations o_i for $1 \leqslant i \leqslant n$, in Effective we would instead write a program p :: Prog $[o_1, \cdots, o_n]$ a and handle it with $h_1 \triangleright \cdots \triangleright h_n$ where each h_i has input effect o_i and no output effects. Even with the non-staged interface, Effective incurs only $O(n)$ overhead in total for composing the handlers (each $\triangleright$ here is $O(1)$) and $O(1)$ overhead for every operation call in p.

Benchmarking. We have used a benchmark suite to compare the performance of Effective with some other Haskell implementations of effect handlers. This suite contains a few simple effectful programs, such as counting down from 10000 to 0 using a mutable state of integers, handled using 1, 11, or 101 layers of handlers. The detailed setup and analysis of the experiment results can be found in the supplementary material `effective-bench/README.md`. The main observations are that non-staged Effective performs reasonably well compared to other libraries and has the distinctive advantage of being insensitive to the number of layers of handlers (thanks to handler fusion). Lightly staged Effective does not show performance improvement in this benchmark because handler combinators are the performance bottleneck in the tests. Fully staged Effective consistently performs very well, being the fastest implementation in the majority of the tests.

Limitations. A limitation of our proposal is that our fusion-based handler combinators are only applicable to back-to-back nested handlers **handle** {**handle** {$\cdots$} **with** h_2} **with** h_1. If handlers and programs are interleaved, such as in **handle** {**do** $x \leftarrow$ (**handle** {$\cdots$} with h_2); op_1} **with** h_1, our handler combinators cannot be used directly. This limitation can also be viewed as a design choice: Effective is inherently a deep-handler library. The intended programming style is to write effectful programs and handlers separately, and apply handlers to programs only in the final step,

as opposed to the programming style encouraged by shallow handlers, where effectful programs and handlers are interleaved. Separating programs and handlers makes reasoning easier, at the cost of some expressiveness. However, this cost of expressiveness of separating programs and handlers is partly compensated by supporting higher-order operations, because the need of interleaving handlers and effectful programs most of the time is for delimiting scopes of operations.

The compile-time performance of Effective is currently a bigger problem for scaling up. A long-known issue of GHC is that it generates quadratically sized internal representation due to recording *every step* of type-family computation and instance resolution, an issue still under investigation after being raised 12 years ago [Izbicki 2013]. This affects Effective because we use type-level lists for tracking effects and inductive instance resolution for $eff \in effs$. In fact, this compilation issue also affects runtime performance, because when the size of the internal representation gets too big, GHC's simplifier stops inlining. We have implemented a GHC type-checking plugin as a half-working stopgap measure until a proper fix to this GHC issue is in place.

Related Work. Our work can be reviewed as a concrete implementation of the categorical foundation of higher-order algebraic effect previously proposed by Yang and Wu [2023], which reconciles the effect handler approach and the monad transformer approach to effectful programming. Yang and Wu [2026] propose a core calculus for higher-order effects and study its meta-theoretic properties. Their calculus is comparable to the foundational layer of Effective of Prog and Algebra. Compared to the work of Yang and Wu [2023, 2026], the present paper focuses on user-friendliness and efficiency, via the handler combinator language and two staging mechanisms.

Our design of Effective resembles the work of Schuster et al. [2020] in a few aspects. Firstly, our impredicative-encoding of programs Prog takes algebras as arguments, resembling the capability-passing style of Schuster et al. [2020]. Secondly, both their work and ours employ staging for additional performance improvement. Their staging mechanism roughly sits in between our light staging and full staging: compared to our light staging, theirs additionally inlines handler code into effectful programs. Compared to full staging, their staging mechanism does not evaluate effectful operations at compile time but only inline them.

Trifanov and Schrijvers [2026] use effect handlers for constraint programming, where searching strategies become handlers of a few primitive search instructions. For performance, they also use Template Haskell to stage the composition of search strategies. In comparison, their work is a domain-specific proposal while Effective is a general framework. We believe that their technique can be implemented as an instance of our (staged) handler fusion combinators.

Xie and Leijen [2021] implement effect handlers using a multi-prompt delimited control monad and evidence passing. Their monad Eff e a of effectful programs is Context $e \to$ Ctl e a where Context e is a vector of handler evidence (which handler clauses are stored in) and Ctl is a delimited control monad. Both this Eff and our Prog have operations (evidence/algebra) passed to programs as an argument which can be efficiently looked up at runtime. However, Prog does not involve delimited control, which is possible because we demand all carriers to have a monadic structure. In Xie and Leijen's [2021] proposal, although finding the corresponding operation clause when invoking an operation can be done in $O(1)$ time, capturing the continuation still needs linear time to 'bubble up' until the corresponding prompt (unless the operation clause is tail resumptive). Our fusion-based handler combinator might be useful for optimising this bubbling-up process.

Conclusion. In this paper we have shown the design and engineering of an implementation of handlers of higher-order effects based on two core ideas: handler combinators and staging. For future work, it is worthwhile to investigate the issue of quadratic-time compilation of type families in GHC, which will benefit our implementation greatly. Another direction is to design and implement 'stage polymorphism' for GHC, which will reduce code duplication and syntactic noise in

our staging mechanisms. Finally, it is worthwhile to study the equational properties of our handler combinators, which will be useful for reasoning about handlers in a more high-level way.

REFERENCES

Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL, Article 62 (2023), 31 pages. https://doi.org/10.1145/3571255

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* Volume 10, Issue 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

Roger Bosman, Birthe van den Berg, Wenhao Tang, and Tom Schrijvers. 2024. A Calculus for Scoped Effects & Handlers. *Logical Methods in Computer Science* Volume 20, Issue 4, Article 17 (2024). https://doi.org/10.46298/lmcs-20(4:17)2024

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems* 18, 6 (Nov. 1996), 730–751. https://doi.org/10.1145/236114.236119

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (Feb. 2018), 62–71. https://doi.org/10.1145/3127323

Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL (2024). https://doi.org/10.1145/3632854

Haskell Contributors. 2025. containers: Assorted concrete container types. Hackage package, version 0.8. https://hackage.haskell.org/package/containers Accessed 2026-05-07.

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development (ICFP'16)*. ACM, 15–27. https://doi.org/10.1145/2976022.2976033

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

John Hughes. 2000. Generalising Monads to Arrows. *Science of Computer Programming* 37, 1 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

Mike Izbicki. 2013. GHC Issue #8095: TypeFamilies painfully slow. https://gitlab.haskell.org/ghc/ghc/-/issues/8095. Accessed: 2026-05-25.

Mauro Jaskelioff and Eugenio Moggi. 2010. Monad Transformers as Monoid Transformers. *Theoretical Computer Science* 411 (2010), 4441–4466. https://doi.org/10.1016/j.tcs.2010.09.011

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Gordon Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 349–360. https://doi.org/10.1145/2103656.2103698

Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021. Efficient compilation of algebraic effect handlers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–28. https://doi.org/10.1145/3485479

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/2535838.2535846

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (2015), 94–105. https://doi.org/10.1145/2887747.2804319

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (ICFP05)*. ACM, 192–203. https://doi.org/10.1145/1086365.1086390

András Kovács. 2024. Closure-Free Functional Programming in a Two-Level Type Theory. *Proceedings of the ACM on Programming Languages* 8, ICFP (Aug. 2024), 659–692. https://doi.org/10.1145/3674648

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. 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, 486–499. https://doi.org/10.1145/3009837.3009872

Paul Blain Levy. 2003. *Call-By-Push-Value.* Springer Netherlands. https://doi.org/10.1007/978-94-007-0954-6

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, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM. http://dl.acm.org/citation.cfm?id=3009897

Žiga Lukšič and Matija Pretnar. 2020. Local algebraic effect theories. *Journal of Functional Programming* 30 (2020). https://doi.org/10.1017/s0956796819000212

Robin Milner. 1980. *A Calculus of Communicating Systems*. Springer.

Eugenio Moggi. 1989a. *An Abstract View of Programming Languages*. Technical Report ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.

Eugenio Moggi. 1989b. Computational Lambda-Calculus and Monads. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 14–23. https://doi.org/10.1109/LICS.1989.39155

Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 941–970. https://doi.org/10.1145/3622831

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. 1996. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96 (POPL '96)*. ACM Press, 295–308. https://doi.org/10.1145/237721.237794

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, 809–818. https://doi.org/10.1145/3209108.3209166

Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell (ICFP 2014)*. ACM, 133–144. https://doi.org/10.1145/2633357.2633360

Gordon 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

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/lmcs-9(4:23)2013

Exequiel Rivas and Mauro Jaskelioff. 2019. Monads with merging. (June 2019). https://inria.hal.science/hal-02150199 working paper or preprint.

Exequiel Rivas and Tarmo Uustalu. 2024. Concurrent monads for shared state. In *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming (PPDP 2024)*. ACM, 1–13. https://doi.org/10.1145/3678232.3678249

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.* ICFP (2020). https://doi.org/10.1145/3408975

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (March 2008), 423–436. https://doi.org/10.1017/s0956796808006758

Wenhao Tang and Sam Lindley. 2026. Rows and Capabilities as Modal Effects. *Proceedings of the ACM on Programming Languages* 10, POPL (Jan. 2026), 923–950. https://doi.org/10.1145/3776674

Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (April 2025), 1130–1157. https://doi.org/10.1145/3720476

Alexandru Trifanov and Tom Schrijvers. 2026. Staging Effect Handlers for Modular Search. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '26)*. ACM, 31–44. https://doi.org/10.1145/3779209.3779536

Birthe van den Berg and Tom Schrijvers. 2024. A framework for higher-order effects & handlers. *Sci. Comput. Program.* 234, C (2024), 32 pages. https://doi.org/10.1016/j.scico.2024.103086

Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 126–154. https://doi.org/10.1145/3704841

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403. https://doi.org/10.1007/978-3-540-70594-9_20

Martin Ward. 1994. Language-oriented programming. *Software—Concepts and Tools* 15, 4 (Oct. 1994), 147–161.

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 302–322. https://doi.org/978-3-319-19797-5_15

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell - Haskell '14* (2014), 1–12. https://doi.org/10.1145/2633357.2633358

Ningning Xie, Daniel D. Johnson, Dougal Maclaurin, and Adam Paszke. 2021. Parallel Algebraic Effect Handlers. https://doi.org/10.48550/ARXIV.2110.07493

Ningning Xie and Daan Leijen. 2021. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 1–30. https://doi.org/10.1145/3473576

Zhixuan Yang. 2024. *Structure and Language of Higher-Order Algebraic Effects.* PhD thesis. Imperial College London.

Zhixuan Yang and Nicolas Wu. 2021. Reasoning about Effect Interaction by Fusion. *Proc. ACM Program. Lang.* 5, ICFP, Article 73 (2021), 29 pages. https://doi.org/10.1145/3473578

Zhixuan Yang and Nicolas Wu. 2023. Modular Models of Monoids with Operations. *Proc. ACM Program. Lang.* 7, ICFP, Article 208 (2023), 38 pages. https://doi.org/10.1145/3607850

Zhixuan Yang and Nicolas Wu. 2026. Handling Higher-Order Effectful Operations with Judgemental Monadic Laws. *Proceedings of the ACM on Programming Languages* 10, POPL (Jan. 2026), 1036–1065. https://doi.org/10.1145/3776678