Latent Effects for Reusable Language Components

Birthe van den Berg[1]([✉]) , Tom Schrijvers[1] , Casper Bach Poulsen[2] ,
and Nicolas Wu[3]

[1] KU Leuven, Leuven, Belgium
{birthe.vandenberg,tom.schrijvers}@kuleuven.be
[2] Delft University of Technology, Delft, The Netherlands
c.b.poulsen@tudelft.nl
[3] Imperial College London, London, UK
n.wu@imperial.ac.uk

Abstract. The development of programming languages can be quite complicated and costly. Hence, much effort has been devoted to the modular definition of language features that can be reused in various combinations to define new languages and experiment with their semantics. A notable outcome of these efforts is the algebra-based "datatypes à la carte" (DTC) approach. When combined with algebraic effects, DTC can model a wide range of common language features. Unfortunately, the current state of the art does not cover modular definitions of advanced control-flow mechanisms that defer execution to an appropriate point, such as call-by-name and call-by-need evaluation, as well as (multi-)staging.

This paper defines *latent effects*, a generic class of such control-flow mechanisms. We demonstrate how function abstractions, lazy computations and a MetaML-like staging can all be expressed in a *modular* fashion using latent effects, and how they can be combined in various ways to obtain complex semantics. We provide a full Haskell implementation of our effects and handlers with a range of examples.

Keywords: Effect handlers · Effects · Monads · Modularity · Staging

1 Introduction

Modern programming languages, be they general purpose or domain-specific, can be built in a flexible manner by composing simple, off-the-shelf language components. It is attractive to build languages in this way as it is useful to study language components in isolation. Furthermore, it reduces the cost of developing new and improved programming languages. Indeed, reducing the effort of building languages to the effort of composing off-the-shelf language components for features such as function abstraction, exceptions or mutable state, is likely to enable language designers with limited resources or expertise—e.g., domain

experts—to build their own languages. Providing a modular definition for these advanced language features enables a more widespread use of them, especially in the development of domain-specific languages (DSLs).

To build programming languages from reusable components, we need a framework for defining those components. Two promising techniques for such a framework are *Data Types à la Carte* [23] (DTC) and *Algebraic Effects & Handlers* [18] (AE&H). Using these, language interpreters can be implemented in two steps:

$$Syntax \xrightarrow{denote} Effects \xrightarrow{handle} Result$$

The first step defines a *denote* function that maps the syntax of an object language onto effectful (*monadic* [13]) operations. By implementing *denote* using DTC, we can seamlessly compose *denote* functions from isolated cases for different *Syntax* constructors.

In the second step, *handle* defines the semantics of effectful operations and their effect interactions. Using AE&H allows cases of *denote* to be defined in an effect polymorphic way; i.e., a case maps to a monad that has *at least* the relevant operations, and possibly more. Furthermore, we can define *handle* functions for different effects in isolation, and seamlessly compose them by nesting handlers. Thus, DTC+AE&H provides a powerful framework for composing programming languages from isolated components.

However, not all language fragments have an obvious modular definition in terms of AE&H. Traditional algebraic effects and scoped effects [29] are baking in assumptions about control-flow and data-flow. These assumptions get in the way of expressing a modular semantics for some language features. In particular, control-flow mechanisms that defer execution, such as lambda abstractions with effectful bodies, lazy evaluation strategies, or (multi-)staging, are neither *algebraic* [17] nor scoped. This implies that the only way to implement these mechanisms is by means of sophisticated encodings that are often relatively low-level and non-modular. As such, these control-flow mechanisms present a severe challenge for allowing non-experts to build programming languages by composing off-the-shelf components.

This paper presents a novel form of effects and handlers that addresses this challenge: *latent effects and handlers*. Latent effect handlers defer the running of side effects, such that they are "latent" until handlers for all effects have been applied. This notion of effects can be used to directly define handlers for language components that defer execution, such as lambda abstractions with effectful bodies, lazy evaluation, and (multi-)staging.

After introducing the background (Sect. 2), our main contributions are:

– We introduce *latent effect trees*, implemented using Haskell (Sect. 3).
– We show how to encode lambda abstraction and application in terms of latent effects. We illustrate how this allows us to define lambda abstraction with effectful bodies as a composable language component (Sect. 3).
– We illustrate how to compose languages from reusable components using latent effects by showing how to encode lazy evaluation (call-by-need and call-by-name), and MetaML-like multi-staging (Sect. 4).

– We provide an effect library[1] with syntax and semantics in Haskell for various simple and advanced language features. These features can be used as isolated components to construct languages.

Finally, we discuss related work (Sect. 5) and conclude (Sect. 6). For an extended version of this work, including a library of effects and an elobarion of our two case studies, we refer to van den Berg et al. [2].

2   Background and Motivation

This section summarizes the state of the art in modular language definition by means of DTC and AE&H, discusses the challenges and problems associated with treating lambdas in this setting, and sketches how our new latent effects enable the integration of modular forms of advanced control-flow effects.

2.1   Modular Syntax and Semantics with Data Types à la Carte

Data types à la Carte [23] (DTC) solves the core problem of assembling language definitions from isolated components. It solves this for both syntax and semantics to be composed in a modular way. For an accessible introduction to DTC, we refer to [23]; we summarize the key points here.

Firstly, the abstract syntax of a language is modularized into the syntax of each isolated feature. This is achieved by defining a recursive type $Syntax\ S$ of ASTs in terms of the shape S of its nodes. This shape S can be composed out of shapes $S_1, \ldots, S_n$ of the individual features: $S = S_1 + \ldots + S_n$.

Secondly, the semantic function $denote :: Syntax\ S \rightarrow M$ that maps ASTs $Syntax\ S$ to their meaning M is modularized into the separate syntactic maps of the individual features. This is done by parameterizing the recursive semantic map of the AST with the semantic mapping of the individual nodes, $denote = fold\ denote_S$ where $denote_S :: S\ M \rightarrow M$ is composed out of the semantic maps:

$$denote_{S_1} :: S_1\ M \rightarrow M \qquad\qquad denote_S = denote_{S_1} + \ldots + denote_{S_n}$$
$$denote_{S_n} :: S_n\ M \rightarrow M$$

This modular approach affords great flexibility to quickly assemble a range of different languages and to explore language design: features can be added or removed, and their semantic maps can be changed. We use the DTC approach extensively when defining a library of effects in Sect. 4.3.

Unfortunately, this approach comes with a serious limitation: to be able to combine the semantic maps of different features, they must agree on the semantic domain M. This often prohibits the unanticipated combinations of features, even due to small differences in their semantic domain, such as one feature requiring access to an environment or store that the other feature does not expect.

---

[1] https://github.com/birthevdb/Latent-Effect-and-Handlers.git.

Moggi [13] observed that many of the differences in semantic domains of common language features, usually called (side-)effects, can be captured in the category-theoretical pattern of *monads*. Although not all monads compose, the state of the art in modular monads, *algebraic effects and handlers* (AE&H) [18], is well-aligned with the DTC approach of modular language features.

In fact, AE&H tackles the problem of modularly defining a monad in much the same way DTC tackles the problem of modularly defining a language. Indeed, the API of the monad (its "syntax") is modularized into separate effects; the category-theoretical concept of a free monad plays the role of an abstract syntax of API calls. A modular meaning is assigned to the free monad by means of semantic maps for the separate effects. The key difference with DTC is that the different effects do not all have to be mapped to the same semantics. Instead, by means of "modular carriers" their semantics can be layered [20].

Whereas AE&H is used to define (simpler) language features, we use latent effects and handlers, which we introduce in Sect. 3, to modularly define more complex language features.

2.2   Advanced, Non-modular Control-Flow Effects

The AE&H literature shows how to express a range of different control-flow effects such as exceptions, non-determinism and even call-with-current-continuation. However, as traditional effects and handlers rely on assumptions about data-flow and control-flow, more advanced and complicated control-flow effects are missing, such as call-by-name and call-by-need evaluation or multi-staging. These features typically defer execution and are *non-algebraic*.

An operation is *algebraic* when it meets the following requirements:

1. It can be expressed as $op :: D \to \forall a. M\ a \to ... \to M\ a$ where D represents the non-computational parameters that the operation has, M is a monad with algebraic operations, and each $M\ a$ typed parameter represents a possible continuation of the operation.
2. It satisfies the *algebraicity property*, which states that, no matter which possible continuation m_i we take, the continuation *immediately* transfers control to the current continuation k.

$$(op\ d\ m_1\ ...\ m_n) \ggg k \equiv op\ d\ (m_1 \ggg k)\ ...\ (m_n \ggg k)$$

Although many operations are algebraic, there are many common control-flow operations that are not. For instance, $catch :: \forall a. M\ a \to M\ a \to M\ a$ is an operation that executes the computation in the first argument, and if that computation raises an exception, proceeds by executing the second argument. The *catch* operation is not algebraic; i.e.,

$$(catch\ m_1\ m_2) \ggg k \not\equiv catch\ (m_1 \ggg k)\ (m_2 \ggg k)$$

since exceptions raised during evaluation of k should *not* be handled by m_2.

The lack of support for control-flow effects such as exception handling, motivated the development of *scoped effects* [29]. An operation is *scoped* when it can be expressed as having the following type:

$$op :: D \to \forall a.M\ a \to ... \to M\ a \to \forall b.(a \to M\ b) \to ... \to (a \to M\ b) \to M\ b$$

The universal quantification over a restricts data flow: for a given operation $op\ d\ m_1...m_n\ k_1...k_m$, it is *only* the possible continuations $k_1...k_m$ that can inspect values yielded by computations m_i. The allowable destinations of data produced by the computation are restricted to those determined by the operation. The *catch* operation is compatible with this restriction: it can be implemented as a scoped operation $catch' :: \forall a.M\ a \to M\ a \to \forall b.(a \to M\ b) \to M\ b$.

However, this pattern does not apply to more advanced control-flow effects for which the data produced by a computation can be used outside of the operation. For example, lambda abstractions delay the execution of computations in the body of a function until the function is applied (or not, depending on dynamic control- and data flow). To support such deferred execution, the return type V in the body of a lambda abstraction $abstr :: String \to M\ V \to M\ V$ is *not* universally quantified. Thus, $abstr$ is not a scoped operation. It is also not algebraic, as the equation $abstr\ x\ m \ggg k \equiv abstr\ x\ (m \ggg k)$ would cause k to be (wrongly) deferred, and could cause free variables in k to be captured. Other control-flow effects, such as call-by-need and call-by-name evaluation and multi-staging annotations for (dynamically) staging and unstaging code, defer execution in a similar way, and are similarly neither scoped nor algebraic.

It is theoretically possible to define the control-flow effects discussed above, by making the control flow of all operations *explicit*; e.g., by writing interpreters in continuation-passing style (CPS). However, the relatively low-level nature of CPS and its non-modular style, make this approach fall short of our goal of composing languages from simple, off-the-shelf components.

2.3   Our Approach: Latent Effects

We provide modular support for advanced control-flow effects such as function abstraction, with its different evaluation semantics, and staging. Our solution generalizes the approach of DTC and AE&H outlined above. In fact, it does not require any changes to the DTC approach for modular abstract syntax trees and modular semantic mapping. It only impacts the second part of the pipeline, replacing AE&H with a more general notion of *latent effects and handlers* (LE&H).

LE&H is based on a different, more sophisticated structure than AE&H's free monad. This structure supports non-atomic operations (e.g., function abstraction, thunking, quoting) that contain or delimit computations whose execution may be deferred. Also, the layered handling is different. The idea is still the same, to replace bit by bit the structure of the tree by its meaning. Yet, while AE&H grows the meaning around the shrinking tree, LE&H grows little "pockets of meaning" around the individual nodes remaining in the tree, and not just

around the root. The latter supports deferred effects because later handlers can still re-arrange the semantic pockets created by earlier handlers.

LE&H are the first to modularly express advanced control-flow effects, such as staging and lambda abstractions, and provide different handlers, e.g., for call-by-name and call-by-need evaluation. Moreover, they combine with existing algebraic effects to express varying semantics for a large range of languages.

3   Latent Effects

This section presents *latent effects*. Latent effects generalize algebraic effects to include control-flow mechanisms that defer computation. As our running example we use lambda abstraction, which—as discussed in Sect. 2.2—is neither an algebraic nor a scoped operation. We show that it can be defined as a latent effect. We start from a non-modular version that we refine in two steps. First we add support for modular signatures, and then support for modular handlers.

3.1   Non-modular Definition of Lambda Abstraction

First we provide a non-modular definition of the lambda abstraction effect.

Monadic Syntax Tree. The type $LC\ v\ a$ is a non-modular monadic syntax tree that supports three primitive operations for a de Bruijn indexed λ-calculus.

```
data LC v a where
    Return :: a → LC v a
    Var    :: Int → (v → LC v a) → LC v a
    App    :: v → v → (v → LC v a) → LC v a
    Abs    :: LC v v → (v → LC v a) → LC v a
```

Here, the v of $LC\ v\ a$ is a value type parameter, and a represents the return type of the computation. Thus *Return x* is a trivial computation that returns x. *Var i k* retrieves the value of type v associated with the ith variable and passes it to the continuation k. The application *App v_1 v_2 k* applies the value v_1 to the value v_2 and passes the result to the continuation. Finally, *Abs e k* builds a closure value out of the function body e and passes it to the continuation.

For example, we can represent the lambda expression $(\lambda x \rightarrow x)\ 1$ as the LC expression *Abs (Var 0 Return) ($\lambda v \rightarrow App\ v\ 1\ Return$)*. This computation constructs an abstraction and passes it to the continuation as a (closure) value v. The continuation applies v to 1 and passes the result to *Return*. The closure retrieves the value of variable with index 0 (i.e., x) and passes it to *Return*.

Handler. The idea of a handler is to map the syntax tree onto its meaning. We illustrate this on the $LC\ v\ a$ type, where we use the type $v = Closure$ for values.

data $Closure_1$ **where**
　　$Clos_1 :: FunPtr_1 \to Env_1 \to Closure_1$
type $Store_1 = [\,LC\ Closure_1\ Closure_1\,]$

type $FunPtr_1 = Int$
type $Env_1 = [\,Closure_1\,]$

A closure contains a function pointer and an environment. The function pointer is an index into a list of deferred computations (i.e., function bodies) that we call the *(resumption) store*. The environment is a list that maps the closure's parameters (which are indexes) onto their values.

Now we are ready to define the handler *handleAbs* as a function that, given an initial environment and store, maps an *LC Closure a* computation onto its meaning, which is a tuple of the result value of type *a* and the final store.

$handleAbs :: Env_1 \to Store_1 \to LC\ Closure_1\ Closure_1 \to (Store_1, Closure_1)$
$handleAbs\ \_\ \ \ \ r\ (Return\ x)\ \ \ \ = (r, x)$
$handleAbs\ env\ r\ (Var\ n\ \ \ \ \ k) = handleAbs\ env\ r\ (k\ (env\ !!\ n))$
$handleAbs\ env\ r\ (App\ v_1\ v_2\ k) = handleAbs\ env\ r'\ (k\ v)$
　　where $(Clos_1\ fp\ env') = v_1$
　　　　　$(r', v)\ \ \ \ \ \ \ \ \ \ \ \ = handleAbs\ (v_2 : env')\ r\ (r\ !!\ fp)$
$handleAbs\ env\ r\ (Abs\ e\ \ \ \ \ \ k) = handleAbs\ env\ r'\ (k\ v)$
　　where $v\ =\ Clos_1\ (length\ r)\ env$
　　　　　$r' = r + [\,e\,]$

First, the leaf case of the handler returns the value in that leaf, supplemented with the resumption store. Next, the variable case looks up the variable in the environment and passes it to the continuation. Then, the application case unpacks the closure, retrieves the corresponding function body from the resumption store and applies it to the extended environment and store. The resulting value is passed to the continuation. Finally, the abstraction case adds the function body to the resumption store, creates a closure that indexes this new entry, and calls the continuation on this closure value.

3.2　Modular Latent Effect Signatures and Trees, Naively

We now modularize the definition of *LC* by separating the recursive structure of the monadic syntax from the node shapes of the *Var*, *App* and *Abs* operations.

Latent Effect Signature. We call the node shapes the latent effect signature. In this case, it is called *Abstracting v* with *v* the type of values.

data $Abstracting\ v :: * \to (* \to *) \to *$**where**
　　$Var' :: Int \to\ \ \ \ \ Abstracting\ v\ v\ NoSub$
　　$App' :: v \to v \to Abstracting\ v\ v\ NoSub$
　　$Abs' ::\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ Abstracting\ v\ v\ (OneSub\ v)$

Besides its first parameter v, the type *Abstracting v p c* is indexed by two further parameters: The parameter p is the return type of the primitive operations; this is the type of value they pass to their continuation. As all three primitive operations return a value of type v, we have that $p = v$. The parameter $c :: * \to *$ captures the number and result type of the subcomputations. As *Var′* and *App′* have no subcomputations, they use the type $c = NoSub$ to indicate that. However, *Abs′* has a subcomputation and it indicates this with $c = OneSub\ v$. This subcomputation is the body of the function abstraction, whose return type is v. Hence, *OneSub v* has one constructor *One :: OneSub v v*.

data $NoSub :: * \to *$**where**   **data** $OneSub\ v :: * \to *$**where**
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad One :: OneSub\ v\ v$

Latent Effect Tree, Version 1. The type $Tree_1\ \sigma\ a$ extends a latent effect signature σ into a recursive syntactic structure that is a monad in a.

> **data** $Tree_1\ (\sigma :: * \to (* \to *) \to *)\ a$ **where**
> $\quad Leaf_1\ :: a \to Tree_1\ \sigma\ a$
> $\quad Node_1 :: \sigma\ p\ c \to (\forall x.c\ x \to Tree_1\ \sigma\ x) \to (p \to Tree_1\ \sigma\ a) \to Tree_1\ \sigma\ a$

The $Leaf_1$ constructor is trivial; $Leaf_1\ x$ returns a pure computation with result x. The internal nodes are of the form *Node op sub k* where the fields have the following meaning. The first, *op*, identifies what primitive operation the node represents. Next, *sub* is a function that, in case of a non-atomic primitive, selects the subcomputations of the node. Finally, k is the continuation of further operations to perform after the current one.

Some notable characteristics are as follows:

– Every operation has a result type p that is made available to its continuation, and a number of subcomputations c. To model these two, the signature of an operation $op :: \sigma\ p\ c$ is parameterized by p and c.
– The function *sub* has type $\forall x.c\ x \to Tree_1\ \sigma\ x$. The input of type $c\ x$ determines what subcomputation to select; the parameter x indicates the result type of that subcomputation.
– Likewise, continuations take as input the operation's output value (p).
– The $Tree_1$ data type is monadic, with a similar notion of return and bind as the *free monad* [23]:

> **instance** $Monad\ (Tree_1\ \sigma)$ **where**
> $\quad return = Leaf_1$
> $\quad (Leaf_1\ x) \ggg f = f\ x$
> $\quad (Node_1\ op\ sub\ k) \ggg f = Node_1\ op\ sub\ (\lambda x \to k\ x \ggg f)$

A monadic binding $t \ggg f$ thus "concatenates" the tree in f to the leaf positions in the continuation (only) of t.

$$var :: (Abstracting\ v < \sigma) \Rightarrow Int \rightarrow Tree_1\ \sigma\ v$$
$$var\ n = Node_1\ (injSig\ (Var'\ n))\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ )\ Leaf_1$$
$$app :: (Abstracting\ v < \sigma) \Rightarrow v \rightarrow v \rightarrow Tree_1\ \sigma\ v$$
$$app\ v_1\ v_2 = Node_1\ (injSig\ (App'\ v_1\ v_2))\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ )\ Leaf_1$$
$$abs :: (Abstracting\ v < \sigma) \Rightarrow Tree_1\ \sigma\ v \rightarrow Tree_1\ \sigma\ v$$
$$abs\ t\ = Node_1\ (injSig\ Abs')\ (\lambda One \rightarrow t)\ Leaf_1$$

Fig. 1. The modular constructor functions of the *Abstracting* effect. These functions all fix the continuation to $Leaf_1$, which can easily be replaced by an arbitrary continuation k using the $\gg\!\!=$ operator of $Tree_1$'s monad instance.

We can emulate the non-modular type $LC\ v\ a$ with $LC'\ v\ a$.

type $LC'\ v\ a = Tree_1\ (Abstracting\ v)\ a$

The corresponding representation for LC's *Return* constructor is $Leaf_1$. The *Var* constructor is represented with a $Node_1$.

$$var_1\quad :: Int \rightarrow (v \rightarrow LC'\ v\ a) \rightarrow LC'\ v\ a$$
$$var_1\ i\ k = Node_1\ (Var'\ i)\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ )\ k$$

This is a *Var' i* node. As there are no subcomputations, there are no branches in the pattern match in the selection function on the value x of the empty type *NoSub*. Lastly, the continuation k receives the value produced by the operation.

The encodings of the two other operations are similar. One notable aspect of abs_1 is that it does have one subcomputation. Hence, the selection function matches on the *One* constructor and returns t:

$$app_1 :: v \rightarrow v \rightarrow (v \rightarrow LC'\ v\ a) \rightarrow LC'\ v\ a$$
$$app_1\ v_1\ v_2\ k = Node_1\ (App'\ v_1\ v_2)\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ )\ k$$
$$abs_1 :: LC'\ v\ v \rightarrow (v \rightarrow LC'\ v\ a) \rightarrow LC'\ v\ a$$
$$abs_1\ t\ k = Node_1\ Abs'\ (\lambda One \rightarrow t)\ k$$

Modular Tree Constructors. We can create modular constructors for latent effect operations, similarly to how DTC admits modular syntax constructors [23]. To this end, we use a co-product operator $+$ that combines latent effect signatures, and a subtyping relation $\sigma_1 < \sigma_2$ with an associated injection function, $injSig :: \sigma_1\ p\ c \rightarrow \sigma_2\ p\ c$. Using these, we can implement the modular constructor functions in Fig. 1, that allow combining *Abstracting* with other latent and algebraic effects. The subtyping requirements in the type signatures are automatically inferrable by type class instance resolution in Haskell. The implementation details of $+$ and $<$ are given in DTC [23] or Appendix A of van den Berg et al. [2].

We can now use these modular constructors to implement denotation function cases. We can also write programs using the constructors directly. For example, the following program with a lambda abstraction with an effectful body:

$prog :: \forall v.Num\ v \Rightarrow Tree_1\ (Mutating\ v + Abstracting\ v + Ending)\ v$
$prog = \textbf{do}$
      $put\ (1 :: v)$
      $f \leftarrow abs\ (\textbf{do}\ m \leftarrow var\ 0; (n :: v) \leftarrow get; return\ (m + n))$
      $put\ (2 :: v)$
      $app\ f\ (3 :: v)$

The body of the function abstraction increments the function argument value (de Bruijn index 0) by the n yielded by the get operation. The signature of the program tree is the co-product of three signatures: (1) $Mutating\ V$ for mutable state; (2) function abstractions $Abstracting\ V$; and (3) the empty signature $Ending$, which provides no operations and serves as the base case. The $Mutating\ V$ effect recasts the traditional algebraic state effect as a latent effect, and has two operations, get and put. Observe that $prog$ is essentially an AST, with multiple possible interpretations. If $Mutating\ V$ is dynamic (runtime) state, then $prog$ evaluates to $3 + 2 = 5$. However, if it is for macro bindings that are expanded *statically*, then the get in the body of the abstraction is evaluated under the state at the "definition site" of the lambda, and $prog$ evaluates to $3 + 1 = 4$. Next, we show how handlers can map the $prog$ syntax tree to different semantics.

3.3   Trees with Support for Modular Handlers

In the case of a modularly composed signature $\sigma = \sigma_1 + ... + \sigma_n + Ending$, the idea is to compose the handler function from individual handlers for the different components of the signature $h = hEnd \circ h_n \circ ... \circ h_1$. The type of each handler would be $h_i :: \forall \sigma. Tree_1\ (\sigma_i + \sigma)\ a \rightarrow Tree_1\ \sigma\ (L_i\ a)$. Hence, it is polymorphic in the remaining part of the signature and preserves those nodes in the resulting tree. It only replaces the nodes of σ_i with their meaning, which is given in the form of a functor L_i that decorates the result type a.

   Unfortunately, our $Tree_1$ type and, in particular, the type of its $Node_1$ constructor, needs some further refinement to fully support this idea. Indeed, if the signature is $\sigma_i + \sigma$, and we apply h_i to all the recursive occurrences of $Tree\ (\sigma_i + \sigma)$ in a σ-node $Node_1\ (Inr'\ op)\ sub\ k$, we get:

Before applying h_i	After applying h_i
$Inr'\ op :: (\sigma_i + \sigma)\ p\ c$	$op\ \ \ \ \ \ :: \sigma\ p\ c$
$sub\ \ \ \ :: \forall x.c\ x \rightarrow Tree_1\ (\sigma_i + \sigma)\ x$	$h_i \circ sub :: \forall x.c\ x \rightarrow Tree_1\ \sigma\ (\ L_i\ x\ )$
$k\ \ \ \ \ \ :: p \rightarrow Tree_1\ (\sigma_i + \sigma)\ a$	$h_i \circ k\ \ :: p \rightarrow Tree_1\ \sigma\ (L_i\ a)$

The resulting fields do not together form a node of type $Tree_1\ \sigma\ (L_i\ a)$ because the highlighted result type of the subcomputations is $(L_i\ x)$ rather than x which the $Node_1$ constructor requires sub to have as return type.

   The problem is that the $Tree_1$ type is oblivious to the *effect functor* that the return type of subcomputations in the tree are decorated by. To solve this problem, we can expose the effect functor decoration in the tree type itself; e.g.,

data $Tree_2$ $(\sigma :: * \to (* \to *) \to *)$ ($l :: * \to *$) a **where**
  $Leaf_2$ $:: a \to Tree_2\ \sigma\ l\ a$
  $Node_2 :: \sigma\ p\ c \to (\forall x.c\ x \to Tree_2\ \sigma\ l\ (\ l\ x))$
        $\to (\ l\ p \to Tree_2\ \sigma\ l\ a) \to Tree_2\ \sigma\ l\ a$

But the $Tree_2$ type requires effect handlers to be applied to subcomputations *immediately*. Motivated by modeling constructs that defer computation, we generalize the type further by parameterizing subcomputations by the effect functor state, and making each node "remember" the effect state (the *latent effects*):

data $Tree$ $(\sigma :: * \to (* \to *) \to *)$ $(l :: * \to *)$ a **where**
  $Leaf$ $:: a \to Tree\ \sigma\ l\ a$
  $Node :: \sigma\ p\ c \to$ $l\ ()$ $\to (\forall x.c\ x \to$ $l\ ()$ $\to Tree\ \sigma\ l\ (l\ x))$
        $\to (l\ p \to Tree\ \sigma\ l\ a) \to Tree\ \sigma\ l\ a$

In Sect. 3.4 we discuss how *Tree* supports deferring computation. *Tree* is a monad with a return and bind defined similarly to the ones for $Tree_1$ in Sect. 3.2. We can also define modular tree constructors using similar techniques as in Sect. 3.2. For instance, using *Tree* instead of $Tree_1$, the type of *prog* from Sect. 3.2 becomes:

$prog :: Tree\ (Mutating\ V + Abstracting\ V + Ending)\ Id\ V$

Here, the *Id* functor models the absence of latent effects in the tree. The type V represents a concrete value type.

Example. Figure 2 shows how the type of the *prog* tree evolves when applying successive handlers for the three parts of the signature:

$(hEnd \circ hAbs\ [\ ]\ [\ ] \circ hMut\ 0)\ prog$

First, we run the modular handler for mutable state *Mutating s*, which has type:

$hMut :: Functor\ l \Rightarrow s \to Tree\ (Mutating\ s + \sigma)\ l\ a \to Tree\ \sigma\ (StateL\ s\ l)\ (s, a)$

Given an initial state of type s, this handler transforms a tree into another tree. The signature of the tree evolves from *Mutating s* $+ \sigma$ to σ because the handler interprets the mutable state, but not the other effects. Also, the latent effect functor evolves from l (the latent effects already present) to *StateL s l*, which augments l with the value of the intermediate state.

newtype $StateL\ s\ l\ a = StateL\ \{\ unStateL :: (s, l\ a)\}$

The result type evolves from a to (s, a), which augments it with the final state.

Second, the handler for *Abstracting V* behaves similarly to *hMut*, removing itself from the signature and growing the latent effect functor. Finally, *hEnd* handles the *Ending* base case. It takes a tree with an empty signature, which thus necessarily only contains a leaf, and extracts the final result out of it.

$$hEnd :: Tree\ Ending\ l\ a \rightarrow a$$
$$hEnd\ (Leaf\ x) = x$$

The remainder of this section illustrates how modular handlers are implemented.

$$Tree\ (\underline{Mutating\ Int} + Abstracting\ V + Ending)\ Id\ V$$
$$\downarrow hMut\ 0$$
$$Tree\ (\underline{Abstracting\ V} + Ending)\ (\ StateL\ Int\ Id)\ (Int,\ V)$$
$$\downarrow hAbs\ []\ []$$
$$Tree\ (\underline{Ending})\ (\ StateL\ (Store\ Ending\ (StateL\ Int\ Id)\ V)\ (StateL\ Int\ Id)$$
$$(\ Store\ Ending\ (StateL\ Int\ Id)\ V\ ,(Int,\ V))$$
$$\downarrow hEnd$$
$$(Store\ Ending\ (StateL\ Int\ Id)\ V, (Int,\ V))$$

Fig. 2. The type of *prog* after successive handling steps.

3.4   Example: Two Modular Handlers for Function Abstractions

We implement two different modular handlers for the operations in *Abstracting*, which illustrate (1) how latent effects let us write handlers for function abstraction, and (2) the kind of fine-grained control the handlers provide. The first handler we consider evaluates the body of a function abstraction using the latent effects of its *call site*. Hence, the evaluation of side-effectful operations is postponed until the function is applied. The second handler evaluates the body of the function abstraction using the latent effects of its *definition site*. This immediately enacts the latent effects introduced by previously-applied handlers.

Modular Closure Values. A concern that arises when we step away from the earlier non-modular handler for *Abstracting* is *reuse*. Notably, in a modular setting we want to allow reuse of both handlers with different notions of values. For that reason, they are parameterized in the type of values v. This type may comprise various shapes of values; all the function abstraction handlers require is that closures are one possible shape of value.

To express this requirement, we introduce another type class $v_1 <: v_2$ for subtyping, this time at kind $*$, which witnesses (1) that any v_1 can be "upcast" to the type v_2; and (2) that a v_2 can be "downcast" to type

v_1. The latter may fail, but the former does not. The minimal requirement for lambda abstractions is that the value type includes closure values.

class $v_1 <: v_2$ **where**
$\quad inj_v :: v_1 \rightarrow v_2$
$\quad proj_v :: v_2 \rightarrow Maybe\ v_1$

data *Closure v* **where**
$\quad Clos :: FunPtr \rightarrow Env\ v \rightarrow Closure\ v$
type *Env v* = $[\,v\,]$

In the modular setting, the types *Closure v* and *Env v*, of respectively closures and value environments, are parameterized in the type of values used.

Modular Resumption Store. Recall that the resumption store keeps track of the function bodies whose execution has been deferred; i.e., it is a list of resumptions. In the modular setting, the type of resumptions is parametric in the specific type of signature, latent effect functor and value type.

type *Store* $\sigma\ l\ v = [\,R\ \sigma\ l\ v\,]$

Moreover, depending on whether we want to handle latent effects on the call site or definition site, the definition of a resumption differs.

A resumption of a call-site effect is a function that takes an $l\ ()$ input, which is the latent effect context of the call site where the resumption is evaluated.

type $R_{CS}\ \sigma\ l\ v = l\ () \rightarrow Tree\ (Abstracting\ v + \sigma)\ l\ (l\ v)$

The resumptions of a definition-site effect store are trees instead of functions that produce trees. Indeed, they have no dependency on the latent effects of the call site. Instead, they have been fully determined by the definition site.

type $R_{DS}\ \sigma\ l\ v = Tree\ (Abstracting\ v + \sigma)\ l\ (l\ v)$

Although the resumption store makes the handlers verbose, it is a more modular solution than storing *Tree*s in values.

Modular Handlers. Figure 3 shows the modular handler $hAbs_{CS}$ that uses the call-site latent effects when executing a function body[2].

Compared to the non-modular definition, there are several differences. Firstly, the handler only interprets part of the work and thus returns the remaining tree instead. Hence, the *Leaf* case now returns a new leaf, and the other cases use monadic **do**-notation to build a new tree. Secondly, because the signature is a composition and the resulting value type is too, the pattern matching on operations involves the Inl' and Inr' tags of the $+$ co-product. The pattern matching and construction of values involves inj_v and $proj_v$ calls for the same reason. Thirdly, the latent effects now matter and need to be properly threaded in all the operation cases. Finally, there is an additional operation case, to handle

---

[2] The function (<$) is short for *fmap ∘ const*.

unknown operations from the remaining part of the signature by "forwarding" them, i.e., copying them to the resulting tree for later handling.

As discussed, in a modular setting a second handler is possible: one that uses the latent effects of the definition site for function bodies rather than their call site. This definition-site handler, $hAbs_{DS}$, looks much like its sibling. The key difference is the type of resumptions, which affects the code in two places (highlighted in gray). Firstly, the abstraction case applies the subtree function to the latent effect of the definition site instead of deferring the application (*st One l* instead of *st One*). Dually, the application case does not have to apply the resumption to the call-site latent effect ($r \mathbin{!!} p$ instead of $(r \mathbin{!!} p)\ l$).

$$hAbs_{CS} :: (Closure\ v <: v,\ Functor\ l)$$
$$\Rightarrow Env\ v \to Store\ \sigma\ l\ v \to Tree\ (Abstracting\ v + \sigma)\ l\ a$$
$$\to Tree\ \sigma\ (StateL\ (Store\ \sigma\ l\ v)\ l)\ (Store\ \sigma\ l\ v, a)$$

$hAbs_{CS}\ \_\ \ r\ (Leaf\ x) \qquad\qquad = Leaf\ (r, x)$
$hAbs_{CS}\ nv\ r\ (Node\ (Inl'\ Abs')\ l\ st\ k) = \textbf{do let}\ v\ =\ inj_v\ (Clos\ (length\ r)\ nv)$
$\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ r' = r \mathbin{+\!\!\!+} [\ st\ One\ ]$
$\qquad\qquad\qquad\qquad\qquad\qquad hAbs_{CS}\ nv\ r'\ (k\ (v \mathbin{<\!\!\$}\ l))$
$hAbs_{CS}\ nv\ r\ (Node\ (Inl'\ (App'\ v_1\ v_2))\ l\ \_\ k) = \textbf{case}\ proj_v\ v_1\ \textbf{of}$
$\quad Just\ (Clos\ fp\ nv') \to \textbf{do}\ (r', v) \leftarrow hAbs_{CS}\ (v_2 : nv')\ r\ (\ (r \mathbin{!!} fp)\ l\ )$
$\qquad\qquad\qquad\qquad\qquad hAbs_{CS}\ nv\ r'\ (k\ v)$
$\quad Nothing \qquad\qquad \to error\ \texttt{"application error"}$
$hAbs_{CS}\ nv\ r\ (Node\ (Inl'\ (Var'\ n))\ l\ \_\ k) \quad = hAbs_{CS}\ nv\ r\ (k\ ((nv \mathbin{!!} n) \mathbin{<\!\!\$}\ l))$
$hAbs_{CS}\ nv\ r\ (Node\ (Inr'\ op) \qquad l\ st\ k) \quad = Node\ op\ (StateL\ (r, l))$
$\quad (\lambda c\ (StateL\ (r', l\ )) \to StateL \mathbin{<\!\!\$\!\!>} hAbs_{CS}\ nv\ r'\ (st\ c\ l))$
$\quad (\lambda\ \ (StateL\ (r', lv)) \to hAbs_{CS}\ nv\ r'\ (k\ lv))$

Fig. 3. Modular call-site abstraction handler. The gray highlights indicate the places where it differs from a definition-site handler.

Example. With the abstraction handlers in place, let us revisit the *prog* example. We run the handlers with default initial values, i.e., 0 for the state, the empty variable environment and the empty resumption store. When using the call-site abstraction handler after the state handler, the function body uses the value of the state that was written right before its invocation.

$\quad > example_{CS} = inspect\ \$\ hEnd\ \$\ hAbs_{CS}\ [\,]\ [\,]\ \$\ hMut\ 0\ prog$
$\quad 5$

If we use the definition-site handler instead, the function body uses the state value that was written right before the abstraction was created.

$\quad > example_{DS} = inspect\ \$\ hEnd\ \$\ hAbs_{DS}\ [\,]\ [\,]\ \$\ hMut\ 0\ prog$
$\quad 4$

4   Case Studies

This section reports on two advanced control-flow features implemented using this library: *call-by-need lambdas* (Sect. 4.1) and *multi-staging* (Sect. 4.2); and on a case study implementation of a library with a range of modular effects (Sect. 4.3). For the source code of these case studies, we refer to the implementation available at https://github.com/birthevdb/Latent-Effect-and-Handlers.git.

4.1   Call-by-Need Evaluation

We have implemented two different evaluation strategies, call-by-need (lazy) and call-by-value (CBV), for lambdas by using different latent effect handlers. Our approach is inspired by Levy's call-by-push-value [10], which can express both strategies. We summarize here; Appendix B of van den Berg et al. [2] has all the details.

Call-by-need evaluation lazily delays the evaluation of argument expressions of function applications, and uses *memoization* to ensure that evaluation only happens once for delayed expressions. We build a lazy semantics for function abstractions out of three primitive effects:

1. The *Reading* effect corresponds to the well-known *reader monad* from the Haskell monad transformer library [11].
2. The *Suspending* effect delays the evaluation of function bodies, without memoizing the result of the evaluation of the delayed subtrees.
3. The *Thunking* effect delays the evaluation of argument expressions of function applications, memoizing the result of forcing a thunked computation.

The definition of these effects and their handlers can be found in the effect library of Appendix A of van den Berg et al. [2]. Using these effects, we define three operations for lazy evaluation (abs_{lazy}, var_{lazy}, and app_{lazy}). Lambda abstraction suspends the body of a lambda, and pairs a pointer to the suspension with the environment that the thunk should be evaluated under. The var_{lazy} and app_{lazy} functions memoize and recall argument values (possibly by forcing the evaluation of a thunked computation), and evaluate the body of a lambda. Application evaluates the first argument to a function value, and memoizes the second argument, which is placed in the current environment. Then, the function body is executed.

The following example program evaluates to 0 when using lazy evaluation:

$$prog_{lazy} :: Tree \ (\ Mutating \ V + Reading \ [\,V\,] + Suspending \ V$$
$$+ \ Thunking \ V + Ending) \ Id \ V$$
$$prog_{lazy} = app_{lazy} \ (abs_{lazy} \ get) \ (\textbf{do} \ put \ 42; get)$$

Function application delays the evaluation of *put* in the argument, and is never executed because the function body does not reference its parameter. We can run the program with call-by-need by applying its handlers:

```
run <print "bar";
  1 + ~(print "foo"; <2>)>
```

$letbind\ (seq\ (print\ \text{"foo"})\ (quote\ (num\ 2)))$
$(unquote\ (quote\ (seq\ (print\ \text{"bar"})$
$(add\ (num\ 1)\ (splice\ (var\ 0))))))$

Fig. 4. A MetaML program (left) and its latent effects implementation (right).

$> inspect\ \$\ hEnd\ \$\ hThunk\ [\,]\ \$\ hSuspend\ [\,]\ \$\ hRead\ [\,]\ \$\ hMut\ 0\ prog_{lazy}$
0

The inspect function extracts the final value out of the result that is decorated with the latent effect functor (in this case nested $StateL$'s).

We can also recover a CBV semantics from app_{lazy}, abs_{lazy}, and app_{lazy} by implementing an alternative handler for the *Thunking* effect. This handler eagerly evaluates subtrees and stores their value in a store.

$> inspect\ \$\ hEnd\ \$\ hEager\ [\,]\ \$\ hSuspend\ [\,]\ \$\ hRead\ [\,]\ \$\ hMut\ 0\ prog_{lazy}$
42

This case study demonstrates that modular call-by-need can be implemented by decomposing it into modular, primitive latent effects and handlers. It also shows how *overloading* the handler of the *Thunking* effect provides a means of changing the semantics of a program without touching the program itself.

4.2 Staging

Another advanced control-flow feature that we have implemented with latent effects is multi-staging. By applying effect handlers *before* the handler for the staging effect, we can control which effects should be staged, and which not. The implementation of these staging constructs can be found in Appendix C of van den Berg et al. [2].

Our inspiration are the three constructs of MetaML [25]: (1) *bracket* expressions (<_>) delay execution to a later stage; (2) *escape* expressions (~_) splice a staged expression into another; and (3) *run* expressions (run _) run an expression that has been dynamically generated by bracket and escape expressions.

A key feature of MetaML is that staged code is *statically typed* and *lexically scoped*. The staging constructs that we implement differ in two ways: our staging constructs are untyped, and we provide *two* constructs for splicing code (*push* and *splice*) instead of the single *escape* expression found in MetaML.

We use *push* for writing programs with escape expressions under binders in staged code. The dynamic semantics of *push* creates an environment with "holes" that represent unknown bindings, and *splice* automatically fills in these holes with bindings from the dynamic context of the *splice* expression.

The four staging constructs we implement are thus: (1) *quote*, corresponding to brackets in MetaML; (2) *unquote*, corresponding to run _ in MetaML; and
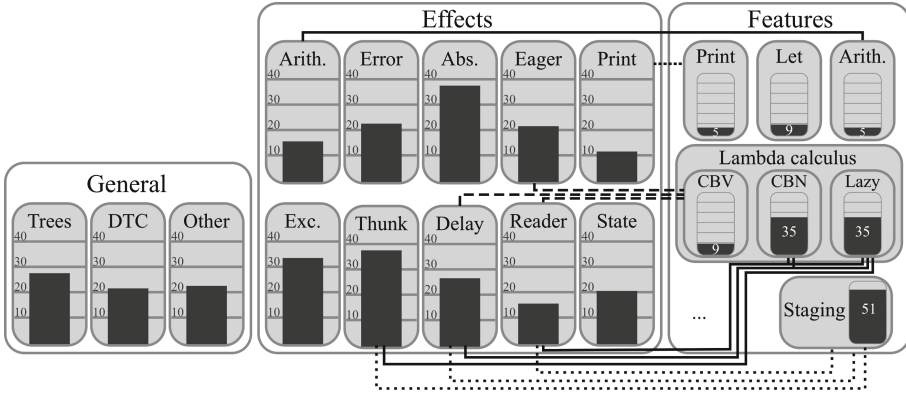
Fig. 5. Effect library with Lines of Code (LoC) per effect.

(3+4) *splice* and *push* for code splicing. The programs in Fig. 4 illustrate the difference in how splicing works. The MetaML program on the left prints the string `"foobar"` and returns the value 3. The program on the right desugars into latent effects. With the appropriate handlers, it gives the same output.

Yet, by switching the order of handlers for the *print* effect and staging, we obtain a different semantics that eagerly handles *print* operations inside *quote*d code. This makes the program on the right print `"barfoo"` instead.

4.3   Library Summary

We have given two examples where latent effects can be modularly composed to form language features. Figure 5 gives an overview of our effect library[3] and how the primitive effects are combined into language features.

The left part shows the general framework code for implementing latent effects, consisting of *Tree*s, the DTC approach and helper definitions (e.g. *Id*, *Void*); the figure also indicates the associated lines of code (LoC). The middle part shows ten different effects and their LoC. Each effect comes with an effect signature, a handler, and smart constructors for their operations. For the detailed implementation of these effects, we refer to the effect library (Appendix A of van den Berg et al. [2]). The right part contains several language features that can be implemented using these effects, with their associated LoC. Each feature comes with its object language syntax and a mapping onto the effects. Each language requires an additional two LoC, to indicate the effects and handlers used and their order. A different order of effects and handlers may give different semantics.

Figure 5 only includes a few language features covered in the paper. However, as we provide ten effects and handlers, they can be modularly composed in different order, using different combinations. In theory, when algebras are fixed,

---

we can define $(10! + 9! + \ldots + 2! + 1!) = 4{,}037{,}913$ semantics, although some compositions may result in the same behaviour. Even more variations are possible, varying the algebra that maps the object language syntax to the effects.

5   Related Work

Modular Semantics and Effects. Modularity has received much attention both at the level of language definitions and of the effects used by those languages. A landmark is the formulation of the expression problem [26], the challenge to modularly extend languages with new features and new interpretation functions. As different language features use different effects, this also requires the modular composition of those effects. Monad transformers [11] are the most prominent approach, including alternative implementations such as Filinksi's layered monads [3] and Jaskelioff's Monatron [7].

Algebraic Effects. Algebraic effects [16] have been proposed as a more structured approach to monads that can also be composed [6]. The subsequent introduction of handlers [18] to deal with exceptions has enabled practical language and library implementations, e.g., [8,9,12]. Schrijvers et al. [20] identified when algebraic effect handlers are modular and related this to a subclass of monad transformers, using the notion of modules [15]. Wu et al. [29] have identified a class of what they call *scoped* effects, which cannot be expressed as algebraic operations. To remedy the situation, they have proposed a practical generalization of algebraic effects. Piróg et al. [14] have put this ad-hoc approach for scoped effects on formal footing in terms of a free monad on a level-indexed category.

Latent Effects. There are many works on specific types of latent effects. For instance, staging is a widely studied area [19,22,24]. Some works have also combined algebraic effects with staging mechanisms [21,27,30]. Yet, we are, to the best of our knowledge, the first to consider staging using effect handlers.

The call-by-push-value calculus of Levy [10] provides primitives for expressing both call-by-name and call-by-value. These have been an inspiration for our modular thunking handler. A more generic work is that of Atkey and Johann [1] on interleaving data and effects to model the incremental production of the data, and on interpreting these with f-and-m algebras.

Various forms of delimited control have been used in the literature to realize sophisticated control mechanisms, such as the simulation of call-by-need by Garcia et al. [5]. Moreover, several works [4] show the interdefinability of conventional algebraic effects and delimited control. A further investigation into the relative expressiveness of latent effects would be interesting.

In future work we would like to demonstrate the performance of latent effects, using the techniques of fusion by Wu and Schrijvers [28].

6   Conclusion

This paper has introduced the notion of latent effects. These extend algebraic effects with the ability to modularly model advanced control-flow mechanisms that can postpone the execution of certain computations and require fine-grained control over the effects inside them. Lambda abstraction, lazy evaluation, and staging were shown to be three prominent instances.

References

1. Atkey, R., Johann, P.: Interleaving data and effects. J. Funct. Program. **25** (2015)
2. van den Berg, B., Schrijvers, T., Bach-Poulsen, C., Wu, N.: Latent effects for reusable language components: extended version (2021)
3. Filinski, A.: Representing layered monads. In: 26th Symposium on Principles of Programming Languages, POPL 1999, pp. 175–188 (1999)
4. Forster, Y., Kammar, O., Lindley, S., Pretnar, M.: On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. J. Funct. Program. **29**, e15 (2019)
5. Garcia, R., Lumsdaine, A., Sabry, A.: Lazy evaluation and delimited control. Log. Methods Comput. Sci. **6**(3) (2010)
6. Hyland, M., Plotkin, G., Power, J.: Combining effects: sum and tensor. Theoret. Comput. Sci. **357**(1–3), 70–99 (2006)
7. Jaskelioff, M.: Monatron: an extensible monad transformer library. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 233–248. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24452-0_13
8. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: 8th Symposium on Haskell, pp. 94–105. ACM (2015)
9. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: 44th Symposium on Principles of Programming Languages, pp. 486–499. ACM (2017)
10. Levy, P.B.: Call-by-push-value: decomposing call-by-value and call-by-name. High. Order Symb. Comput. **19**(4), 377–414 (2006)
11. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Symposium on Principles of Programming Languages, pp. 333–343. ACM (1995)
12. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: 44th Symposium on Principles of Programming Languages. ACM (2017)
13. Moggi, E.: An abstract view of programming languages. Technical report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989
14. Piróg, M., Schrijvers, T., Wu, N., Jaskelioff, M.: Syntax and semantics for operations with scopes. In: Logic in Computer Science, pp. 809–818. ACM (2018)
15. Piróg, M., Wu, N., Gibbons, J.: Modules over monads and their algebras. In: 6th Conference on Algebra and Coalgebra in Computer Science. LIPIcs, vol. 35, pp. 290–303. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)

16. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_24
17. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Appl. Categ. Struct. **11**(1), 69–94 (2003)
18. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 80–94. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_7
19. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: Generative Programming and Component Engineering, pp. 127–136. ACM (2010)
20. Schrijvers, T., Piróg, M., Wu, N., Jaskelioff, M.: Monad transformers and modular algebraic effects: what binds them together. In: 12th International Symposium on Haskell, pp. 98–113. ACM (2019)
21. Schuster, P., Brachthäuser, J.I., Ostermann, K.: Compiling effect handlers in capability-passing style. Proc. ACM Program. Lang. **4**(ICFP), 93:1–93:28 (2020)
22. Sheard, T., Jones, S.L.P.: Template meta-programming for Haskell. ACM SIGPLAN Not. **37**(12), 60–75 (2002)
23. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008)
24. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: PEPM, pp. 203–217 (1997)
25. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1), 211–242 (2000). PEPM 1997
26. Wadler, P.: The expression problem. Mailing list (1998)
27. Wei, G., Bračevac, O., Tan, S., Rompf, T.: Compiling symbolic execution with staging and algebraic effects. Proc. ACM Program. Lang. **4**(OOPSLA) (2020)
28. Wu, N., Schrijvers, T.: Fusion for free. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 302–322. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_15
29. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Symposium on Haskell, pp. 1–12. ACM (2014)
30. Yallop, J.: Staged generic programming. Proc. ACM Program. Lang. **1**(ICFP), 29:1–29:29 (2017)