Staged Effects and Handlers for Modular Languages with Abstraction

Casper Bach Poulsen
c.b.poulsen@tudelft.nl
Delft University of Technology
The Netherlands

Cas van der Rest
c.r.vanderrest@tudelft.nl
Delft University of Technology
The Netherlands

Tom Schrijvers
tom.schrijvers@cs.kuleuven.be
KU Leuven
Belgium

Abstract

This short paper aims to use modular effects to define modular languages with lambda abstraction. We argue that existing approaches to algebraic effects and handlers are not suitable for this challenge. Instead, we propose a new approach that we dub *staged effects and handlers*. We show how to use our approach to define lambda abstraction in a modular way, and discuss open questions.

CCS Concepts: • **Software and its engineering** → *Functional languages*.

Keywords: effect handlers, effects, monads, modularity, semantics

ACM Reference Format:
Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. 2021. Staged Effects and Handlers for Modular Languages with Abstraction. In *PEPM '21.* ACM, New York, NY, USA, 7 pages. https://doi.org/xxx

1 Introduction

This paper considers how to define languages *modularly* in terms of a *compositional* denotation function:

$$[\![\_]\!] \;:\; \mathsf{Expr} \;\rightarrow\; \mathsf{M}\;\mathsf{V}$$

Here, Expr is the type of abstract syntax trees, M is a monad, and V is a value type. By *modular* we mean that one can (i) add new constructors to Expr, and (ii) add new *operations* to M without modifying existing code. By *compositional* we mean that ⟦_⟧ defines the semantics of a complex expression in terms of the semantics of its *recursive sub-trees*. Compositionality is attractive because it gives a semantics that is nice to reason about [21], makes it possible to reuse ⟦_⟧ for different M (for example, we could use ⟦_⟧ to define either a *static* or *dynamic* semantics of Expr), and provides a clear path to the first dimension of modularity (adding constructors to Expr).

Existing work on *data types à la carte* [13, 24] addresses the first dimension of modularity. *Algebraic effect handlers* [18] are a flexible and popular framework that can be used to address the second dimension of modularity. However, we cannot readily define some classes of effects modularly using effect handlers. Wu et al. [26] observe:

> One aspect of handlers that has not received much attention are scoping constructs. Examples of this are abound: we see it in constructions for control flow, such as while loops and conditionals, but we also see this in pruning non-deterministic computations, exception handling, and multi-threading.

There is, however, another aspect of handlers that has not received much attention: *staging constructs*. Examples of interesting staging constructs and applications are abundant in the literature on programming languages (and PEPM in particular) [2, 11, 20, 22, 25, 27]. We focus on one kind of staging construct, namely *lambda abstraction*. An expression λx. e *stages* (postpones) the evaluation of e, and function application *unstages* it. It is hard to fit this kind of staging in existing frameworks for effects and handlers [17, 18, 26]. In particular, it is difficult to implement the following operations using algebraic effects and handlers:

```
abstr : Name → M Val → M Val
apply : Val → Val → M Val
```

In this short paper we address this challenge by proposing a new kind of effect handler: *staged effect handlers*. We use Agda[1] as our meta language, assuming a passing familiarity with dependent types, but not an in-depth knowledge of Agda. The techniques we describe in this paper could also be defined a functional language without dependent types, such as Haskell or Scala. Our motivation for using Agda is a desire to eventually use the framework proposed in this paper to implement *modular* and *intrinsically-typed* language definitions. For this paper, however, we limit ourselves to *simply-typed* languages. An artifact that implements the framework described in this paper is available online:

https://github.com/casvdrest/staged-effects.agda

The paper is structured as follows. § 2 defines three modular language fragments that we use as running examples. Next, § 3 defines "plain" effects and handlers [18] in Agda for only one of the modular fragments. Then § 4 shows that *scoped effects and handlers* [17, 26] provide additional expressiveness, but conclude that scoped effects and handlers

PEPM '21, January 18–19, 2021, Copenhagen, Denmark
2021. ACM ISBN 978-1-4503-XXXX-X/18/06. . . $15.00
https://doi.org/xxx

[1] https://agda.readthedocs.io/

alone are insufficient for defining *staging constructs* (lambda abstraction and application). Finally, in § 5 we propose a new notion of *staged effects* and handlers that lets us define both staging constructs and scoping constructs. § 6 concludes.

2 Compositional Semantics for Languages with lambda abstraction and State

Our goal is to implement a modular and compositional semantics for the following object language:

```
Expr ∋ e ::= var x | abs x e | app e e | let x = e in e
           | get | put e | nat n
```

where x ∈ Name ranges over names and n ∈ ℕ ranges over natural numbers. The semantics we consider is given by a function ⟦_⟧ : Expr → M V, where M is an instance of the following families of monads:

```
record LambdaM (M : Set → Set) (V : Set) : Set where
  field fetch   : Name → M V
        abstr   : Name → M V → M V
        apply   : V → V → M V
        letbind : Name → V → M V → M V
record StateM (M : Set → Set) (S : Set) : Set where
  field get : M S
        put : S → M ⊤
record NatM (M : Set → Set) (V : Set) : Set where
  field nat : ℕ → M V
```

It may seem overly general to have letbind as its own operation, rather than desugaring it into abstr and apply. However, while this paper focuses on the problem of using effect handlers to define an interpreter for the language above, we are ultimately interested in a broader goal: we want to write denotation functions ⟦_⟧ : Expr → M V that use effect handlers to modularly define diverse semantic artifacts such as *modular compilers* [5], *modular abstract interpreters* [3, 12, 19], *modular symbolic executors* [15], and other semantic artifacts. If we desire an M V that defines a *static semantics*, the semantics of lambda binding and let binding may differ (as in Hindley-Milner-Damas polymorphism [4, 10, 16]), and desugaring would be wrong.

In the remainder of this section we show how to define ⟦_⟧ in a way that lets us extend Expr with new constructors without modifying existing code by using *data types à la carte* [24] (DTC). The rest of this paper considers the challenge of extending M with new operations without modifying existing code.

2.1 Modular Syntax

A prerequisite for using DTC to define ⟦_⟧ in a modular way, is that Expr is defined in a modular way. We define a modular data type for Expr following Keuchel and Schrijvers [13] by using *containers* [1] to ensure that our data types are strictly-positive (as Agda requires). A container consists of a *shape*, S, and position, P:

```
record Con : Set₁ where
  constructor _▹_
  field S : Set
        P : S → Set
```

The shape describes the set of available *constructors*, and the position maps each constructor to its corresponding *arity* (i.e., the set of recursive sub-trees). For example, the following container encodes an expression type for the state fragment of our object language (e ::= get | put e):[2]

```
StateExpr = Bool ▹ λ { false → ⊥; true → ⊤ }
```

Here we use a type with two inhabitants (one for put, one for get) as the shape: Bool. Each inhabitant (false and true) is associated with a position: the false case corresponds to get which has no recursive sub-trees, so the set of recursive positions is given by the empty type ⊥. On the other hand, put has a single recursive argument, so the position for true is associated with the unit type ⊤.

We relate container-encoded expressions to Agda types by defining their *semantics* of type Set → Set:

$$
\begin{aligned}
&[\![\_]\!]^c : \mathsf{Con} \to \mathsf{Set} \to \mathsf{Set} \\
&[\![\, S \triangleright P \,]\!]^c\, X = \exists\, \lambda\, (s : S) \to P\, s \to X
\end{aligned}
$$

A container is interpreted as a pair of a constructor s : S and a function P s → X that maps each recursive position of s to an Agda value of type X. To interpret data types with recursive positions we need to take their least fixed-point:

$$
\begin{aligned}
&\mathbf{data}\ \mu\ (C : \mathsf{Con}) : \mathsf{Set}\ \mathbf{where} \\
&\quad \langle\_\rangle : [\![\, C \,]\!]^c\, (\mu\, C) \to \mu\, C
\end{aligned}
$$

Using this fixed-point, expressions in the state fragment are typed by μ StateExpr.

Containers have a well-defined notion of *union*:[3]

$$
\begin{aligned}
&\_\cup\_ : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Con} \\
&(S_1 \triangleright P_1) \cup (S_2 \triangleright P_2) = \\
&\quad S_1 \uplus S_2 \triangleright \lambda\, \{ (\mathsf{inj}_1\, x) \to P_1\, x ; (\mathsf{inj}_2\, y) \to P_2\, y \}
\end{aligned}
$$

Using this union, we can modularly compose StateExpr with the container description of nat expressions (e ::= nat n):

```
NatExpr = ℕ ▹ const ⊥
```

Here, ℕ is the shape (there are as many nat expressions as there are naturals) and const ⊥ says there are no recursive sub-trees. The state+nat fragment of our object language (e ::= get | put e | nat n) is thus given by StateExpr ∪ NatExpr. By similarly encoding the lambda fragment (LamExpr) we can compose Expr from modular syntax fragments:

$$\mathsf{Expr} \simeq \mu\, (\mathsf{LamExpr} \cup \mathsf{StateExpr} \cup \mathsf{NatExpr})$$

[2] λ { _ } is Agda syntax for a pattern matching lambda; ⊥ is the empty type; and ⊤ is the unit type.
[3] X ⊎ Y is the type of a disjoint sum in Agda, whose constructors are $\mathsf{inj}_1 : X \to X \uplus Y$ and $\mathsf{inj}_2 : Y \to X \uplus Y$.

2.2 Modular Semantic Functions

We encode semantic functions for container-encoded expression types as *algebras*, given by the following type alias:

$$\mathsf{C} \Rightarrow \mathsf{A} \triangleq [\![\mathsf{C}]\!]^c\, \mathsf{A} \to \mathsf{A}$$

By *folding* an algebra C $\Rightarrow$ A over a data type μ C we can turn recursive sub-trees into values A:

```
foldᶜ : (C ⇒ A) → μ C → A
foldᶜ f ⟨ s , p ⟩ = f (s , foldᶜ f ∘ p)
```

This use of folds necessitates a *compositional* semantics: *by definition*, algebras encode structurally-recursive (i.e., compositional) functions. The following algebra defines the semantics of StateExpr expressions:[4]

```
algState : ⦃ StateM M V ⦄ → StateExpr ⇒ M V
algState (inj₁ tt , _) = get
algState (inj₂ tt , p) = do v ← p tt; put v; return v
```

Algebras ranging over two different containers C_1 and C_2 can be combined into an algebra ranging over $\mathsf{C}_1 \cup \mathsf{C}_2$ using a function (whose implementation we elide for brevity):

```
_⊙_ : (C₁ ⇒ A) → (C₂ ⇒ A) → C₁ ∪ C₂ ⇒ A
```

Using algebra composition and our monad families, we can compose algebras:

```
alg : ⦃ LambdaM M V ⦄ →
      ⦃ NatM M V     ⦄ →
      ⦃ StateM M V   ⦄ → Expr ⇒ M V
alg = algLam ⊙ algNat ⊙ algState
```

to obtain a denotation function $[\![\_]\!] \simeq \mathsf{fold}^c$ alg.

We have shown how to modularly define expression types and their semantics, by modularly mapping expressions onto a monad families. We have left open the question of how instances of these monad families are defined. Indeed, if we use "standard" monads, way may need to modify the implementation of each monad family when we add new effects. Both monad transformers [14] and the slightly more structured algebraic effects and handlers afford more flexibility. In the rest of this paper we consider how to use algebraic effects and handlers to define the monad family instances for alg above in a way that does not require modifying existing code.

3 Effects and Handlers

We illustrate how to define algebraic effects and handlers in Agda as a *free monad*. The idea is to represent computations as trees of possible sequences of effectful operations. Following Hancock and Setzer [9], the type of such trees (I/O trees) is IO σ A where σ : Con is a *signature* of operations given by a container. Signatures can be freely composed using the _∪_ : Con → Con → Con function from § 2. I/O trees are given by the following data type:

```
data IO (σ : Con) : Set → Set where
  end : A → IO σ A
  cmd : (c : S σ) → (P σ c → IO σ A) → IO σ A
```

The constructor end represents a "pure" computation. The cmd constructor represents an effectful operation whose constructor is given by c : S σ, and whose continuation is parameterized by the *return type* P σ c of the operation. IO trees are monadic with the end constructor as the return of the monad, and with the following notion of bind:

```
_≫=_ : IO σ A → (A → IO σ B) → IO σ B
end x    ≫= k = k x
cmd c p  ≫= k = cmd c (λ x → p x ≫= k)
```

A signature for two stateful operations, ‘get and ‘put, is given by the following data type (defining a set of constructors) and signature:[5]

```
data StateOp (H : Set) : Set where
  ‘get : StateOp H
  ‘put : H → StateOp H

StateSig : Set → Con
S (StateSig H)          = StateOp H
P (StateSig H) ‘get     = H
P (StateSig H) (‘put h) = ⊤
```

Trees with ‘get and ‘put operations are an instance of the StateM record from § 2 by using a generic lift function, defined in terms of a signature subtyping judgment (_ ≪ _) (we elide the implementations of lift and _ ≪ _ for brevity, and refer to our repository for the full details):

```
lift : (σ₁ ≪ σ₂) → (c : S σ₁) → IO σ₂ (P σ₁ c)

StateInst : (StateSig H ≪ σ) → StateM (IO σ) H
get (StateInst w)   = lift w ‘get
put (StateInst w) h = lift w (‘put h)
```

The following effect handler for state operations handles state effects in a manner that is agnostic to what other effects a IO tree may contain:[6]

```
hSt : H → IO (StateSig H ∪ σ) A → IO σ A
hSt _ (end x)                  = end x
hSt h (cmd (inj₁ ‘get) k)      = hSt h (k h)
hSt _ (cmd (inj₁ (‘put h)) k)  = hSt h (k tt)
hSt h (cmd (inj₂ y) k)         = cmd y (hSt h ∘ k)
```

It is equally straightforward to define a handler for the nat operation of the NatM family. We might try to define a handler for the operations in LambdaM as well. However, the monadic arguments of the letbind and lambda operations pose a challenge: the IO type only admits branching over possible continuations. In the term letbind x v m, the subterm m is a *scoped computation* and not a continuation in

[4]Arguments enclosed in double curly braces (i.e., ⦃_⦄) are automatically filled in by Agda using instance resolution.

[5]The StateSig function uses co-patterns to define the values of the fields in the returned Signature. For example, The line S (StateSig H) = … defines the S field of the record StateSig H.

[6]It would have been equally possible to define this handler in terms of a generic fold over IO trees; a so-called deep handler.

the sense that IO supports. In the next section we show how *scoped effects and handlers* [17, 26] let us implement the letbind operation, but argue that both plain and scoped effects and handlers are insufficient for handling lambdas.

4 Scoped Effects and Handlers

The IO trees from the previous section do not support *scoping constructs*. This makes it challenging to define LambdaM's letbind operation in a modular manner. In this section we illustrate how this shortcoming is addressed by *scoped effects and handlers*, due to Wu et al. [26] and Piróg et al. [17]. The encoding of trees with scoped effects shown in this section is equivalent to that of Piróg et al. [17].

4.1 Trees With Scoped Effects

Trees with scoped effects are given by the type Prog σ γ A where σ : Con is the signature for "plain" operations (like the ones in § 3), and γ : Con is the signature of scoping constructs. Trees of operations and scope constructs are given by the following Prog data type:

```
data Prog (σ γ : Con) (A : Set) : Set₁ where
  var   : A → Prog σ γ A
  op    : (c : S σ) → (P σ c → Prog σ γ A) →
          Prog σ γ A
  scope : (g : S γ) → (P γ g → Prog σ γ B) →
          (B → Prog σ γ A) → Prog σ γ A
```

The var and op constructors correspond to the end and cmd constructors of IO (§ 3). The scope constructor represents an occurrence of a scoping construct with a set of scopes (P γ g → Prog σ γ B) and a continuation (B → Prog σ γ A).

Programs are monadic, with the var constructor as the return of the monad, and with the following notion of bind:

```
_≫=_ : Prog σ γ A → (A → Prog σ γ B) → Prog σ γ B
var x        ≫= g = g x
op c k       ≫= g = op c (λ x → k x ≫= g)
scope s sc k ≫= g = scope s sc (λ x → k x ≫= g)
```

4.2 Effect Weaving

A key difference between plain effect handlers and scoped effect handlers is that scoped effect handlers *weave* effects through *both* continuations *and* scopes, as illustrated in the last case of the following handler for 'get and 'put:

```
hSt' : H → Prog (StateSig H ∪ σ) γ A → Prog σ γ (A × S)
hSt' h (var x) = var (x , h)
hSt' h (op (inj₁ 'get) k) = hSt' h (k h)
hSt' _ (op (inj₁ ('put h)) k) = hSt' h (k tt)
hSt' h (op (inj₂ y) k) = op y (hSt' h ∘ k)
hSt' h (scope g sc k) =
  scope g (hSt' h ∘ sc) (λ {(x , h') → hSt' h' (k x)})
```

Note that hSt' coincides with hSt when γ is empty.

4.3 Defining and Handling Let Binding

The scope constructor lets us define effects for variable lookups and let bindins modularly, by means of the following signature definitions:

```
data FetchOp : Set where
  'fetch : Name → FetchOp

FetchSig : Set → Con
S (FetchSig V) = FetchOp
P (FetchSig V) ('fetch x) = V

data LetScope (V : Set) : Set where
  'letbind : Name → V → LetScope V

LetSig : Set → Con
S (LetSig V) = LetScope V
P (LetSig V) ('letbind n v) = ⊤
```

By modeling letbind as a *scoped effect*, we can handle let binding and variable fetching using the following handler for FetchSig and LetScope:[7]

```
Env : Set → Set
Env V = List (Name × V)

hLet : Env V →
         Prog (FetchSig V ∪ σ) (LetSig V ∪ γ) A →
         Prog σ γ (Maybe A)
hLet _ (var x) = var (just x)
hLet E (op (inj₁ ('fetch x)) k) =
  maybe (hLet E ∘ k) (var nothing) (lookup E x)
hLet E (op (inj₂ c) k) = op c (hLet E ∘ k)
hLet E (scope (inj₁ ('letbind n v)) sc k) =
  hLet ((n , v) :: E) (sc tt) ≫=
    maybe (hLet E ∘ k) (var nothing)
hLet E (scope (inj₂ g) sc k) =
  scope g (hLet E ∘ sc) (maybe (hLet E ∘ k) (var nothing))
```

4.4 The Challenges With Handling Lambda

We could try to define a handler for lambda in a similar manner as hLet. Since a lambda scopes the effects that are stored in the body of the function, our only choice is to define lambda as a scoping construct; i.e.:

```
data LamScope : Set where
  'lambda : Name → LamScope

LamSig : Set → Con
S (LamSig V) = LamScope
P (LamSig V) ('lambda n) = V
```

However, it is not obvious how to define a handler for this scoping construct that behaves as we would expect lambdas to behave. Consider the following handler function with a hole ({!!}) in it:

```
hLam : Env V →
         Prog (FetchSig V ∪ σ) (LamSig V ∪ γ) A →
         Prog σ γ (Maybe A)
```

[7] tt is the unit value and maybe is the eliminator of the Maybe type.

```
  -- ...
hLam E (scope (inj₁ (‘lambda n)) sc k) = hLam E (k {!!})
hLam E (scope (inj₂ g) sc k) =
  scope g (hLam E ∘ sc) (maybe (hLam E ∘ k) (var nothing))
```

There are two problems with here. The first problem is that in the application k {!!}, the hole must be filled with a value of some generic type B. However, the continuation k is *supposed* to accept a lambda (closure) value. The root of the issue is that the scope constructor says scopes have a *polymorphic return type*. This polymorphism is essential for weaving effect handlers through scopes, but here it gets in the way.

The second problem is that, by defining lambdas as a scoping construct, *effect handlers will always be applied under lambdas*. For example, the weaving that happens in the last case of hSt' from § 4.2 will cause stateful operations under lambdas to be evaluated *before the function is applied*. For example, consider this program which we would expect to yield 42:

```
prog = do n₀    ← nat 0; put n₀
          closr ← lambda x get
          n₄₂   ← nat 42; put n₄₂
          apply closr n₀
```

If we apply the state handler hSt' above before we apply hLam then weave will eagerly evaluate the get under the lambda, causing the operation to be replaced by the value 0, giving the wrong result: 0 instead of 42!

5 Staged Effects and Handlers

We show how to overcome both the first and the second problem summarized above, by introducing a new Tree type that supports staged effects. This Tree type is based on two ideas. Firstly, instead of requiring scoped computations to always have a *polymorphic type*, the signatures of staged operations fix the return types of each scoped computation. (This addresses the first problem we identified above.) Secondly, instead of requiring that handlers are always fully applied when we weave them through effect scopes, Tree lets us weave *partially-applied* handlers through effect scopes, such that we can, for example, postpone applying a state handler to a store. (This addresses the second problem we identified above.)

5.1 Trees With Staged Effects

Trees with staged effects are given by the type Tree L ζ A, where L : Set $\rightarrow$ Set is a functor representing the set of *latent effects* of nodes in the tree, and ζ : Sig is the signature of operations with staging. ζ signatures are comprised of a pair of a regular signature σ : Con, similar to I/O trees, and a σ-dependent signature ξ : S σ $\rightarrow$ Con which says what the parameter- and return-types are of staged effect scopes. For convenience, the following Sig type combines dependent σ,ξ pairs in a single record type:

```
record Sig : Set₁ where
  field S₁ : Set;        P₁ : S₁ → Set
        S₂ : S₁ → Set; P₂ : ∀ {s₁} → S₂ s₁ → Set
```

Signatures of operations with staging have a straightforward notion of sum _⊕_ and subtyping _ ⊏ _ (whose implementations we elide for brevity), analogous to regular signatures.

Trees with staged effects are given by the following type:

```
data Tree (L : Set → Set) (ζ : Sig) (A : Set) : Set₁ where
  leaf  : A → Tree L ζ A
  node  : (c : S₁ ζ) → L ⊤ →
          ((s₂ : S₂ ζ c) → L ⊤ → Tree L ζ (L (P₂ ζ s₂))) →
          (L (P₁ ζ c) → Tree L ζ A) → Tree L ζ A
```

The arguments of a 'node c l st k' are: (i) a constructor c; (ii) latent effects l; (iii) staged effect scopes st : (s_2 : S_2 ζ c) $\rightarrow$ L $\top$ $\rightarrow$ Tree L ζ (L (P_2 ζ s_2)); and (iv) a continuation expecting a response wrapped in a latent effect context (L (P_1 ζ c)). The latent effects l are a main difference between the Tree type and the Prog type from the previous section: each node in a Tree "remembers" which effects other effect handlers have propagated past the node, effectively *staging* these effects. For example, after applying a state handler, each node in the tree "remembers" which store it should be evaluated under. By parameterizing staged effect scopes by an effect context L $\top$, we can weave handlers through scopes in a way that these handlers are evaluated relative to some "future" effect context. In § 5.3 we illustrate how this lets us propagate handlers for state under lambdas in a way that state operations inside lambda bodies are handled relative to a future store.

Unlike the Prog type from the previous section, the Tree type above does not have separate constructors for "plain" or "scoped" operations. We conjecture that "plain" and "scoped" operations can be defined as special cases of nodes in a Tree.

Trees are monadic, with the leaf constructor as the return of the monad, and with the following notion of bind:

```
_ ≫= _ : Tree L ζ A → (A → Tree L ζ B) → Tree L ζ B
leaf x       ≫= g = g x
node z l st k ≫= g = node z l st (λ x → k x ≫= g)
```

5.2 Effect Staging

Below is the handler for state defined in terms of Tree:[8]

```
hSt” : ⦃ RawFunctor L ⦄ →
       H → Tree L (StateSig H ⊕ ζ) A →
       Tree ((H ×_) ∘ L) ζ (H × A)
hSt” h (leaf x) = leaf (h , x)
hSt” h (node (inj₁ ‘get) l _ k) = hSt” h (k (const h <$> l))
hSt” _ (node (inj₁ (‘put h)) l _ k) = hSt” h (k l)
hSt” h (node (inj₂ c) l st k) =
  node c (h , l)
```

[8]RawFunctor L says that L is a functor, and _<$>_ is the map function of the functor instance. Note that StateSig H : Sig is a straightforward adaptation of the StateSig H : Con from § 3.

```
      (λ { z (h' , l') → hSt” h' (st z l') })
      (λ { (h' , lr) → hSt” h' (k lr) })
```

The ‘get case now enacts the latent effects l of their nodes by injecting the response value into the latent effect context l : L ⊤. The ‘put case also enacts the latent effects by the application of k to l. The last case of hSt” weaves the hSt” handlers through nodes other than StateSig node, by wrapping the latent effects l in the state functor (H ×_), staging the passing of the “current” store h (or perhaps an extension thereof) to the staged effect scope st and continuation k.

5.3 Defining and Handling Let Binding and Lambda

The Tree type lets us define the syntax and handling of the operations in LambdaM in a modular manner. We use the following record type to assert the existence of introduction and elimination functions for closures:

```
record ClosureVal (V : Set) : Set where
  field close  : Name → FunLabel → Env V → V
        isClos : V → Maybe (Name × FunLabel × Env V)
```

Here, FunLabel is a pointer into a “resumption store” comprising the (latently effectful) code of function bodies:

```
Resumptions : (Set → Set) → Sig → Set → Set
Resumptions L σ V =
  List (L ⊤ → Tree L (LamOpSig V ⊕ σ) (L V))
```

The motivation for representing closures and storing them in a store in this way is *modularity*: by using labels to denote function bodies, our notion of value makes no assumptions about what latent effects are in the Trees of function bodies. Only the handler hLam' below needs to know the actual type of function bodies. The handler uses try m f = maybe f (leaf nothing) m for mapping an f : A → Tree L ζ (Maybe B) over an m : Maybe A, and is parameterized by: (i) an environment Env V (for variable binding); (ii) a resumption store (for allocating and dereferencing function values); and (iii) a “fuel” counter [23]. The fuel counter is for ensuring that hLam' terminates (by bottoming out and returning nothing) for diverging functions.

```
hLam' : ⦃ ClosureVal V ⦄ → ⦃ RawFunctor L ⦄ →
        Env V → Resumptions L ζ V → ℕ →
        Tree L (LamOpSig V ⊕ ζ) A →
        Tree (Maybe ∘ (Resumptions L ζ V ×_) ∘ L)
             ζ (Maybe (Resumptions L ζ V × A))
  -- elided: leaf and case for out-of-fuel exception
hLam' E funs (suc m) (node (inj₁ (‘app v₁ v₂)) l _ k) =
  try (isClos v₁) λ { (n , f , E') →
    try (retrieve funs f) (λ r →
      hLam' ((n , v₂) :: E') funs m (r l) ≫=
        flip try (λ { (funs' , lv) →
          hLam' E funs' m (k lv) })) }
hLam' E funs (suc m) (node (inj₁ (‘fetch n)) l _ k) =
  try (lookup E n) (λ v →
    hLam' E funs m (k (const v <$> l)))
hLam' E funs (suc m) (node (inj₁ (‘abs n)) l st k) =
  hLam' E (funs ++ [ st tt ]) m
        (k (const (close n (length funs) E) <$> l))
hLam' E funs (suc m) (node (inj₁ (‘letbind n v)) l st k) =
  hLam' ((n , v) :: E) funs m (st tt l) ≫=
    flip try λ { (funs' , lv) → hLam' E funs' m (k lv) }
hLam' E funs (suc m) (node (inj₂ c) l st k) =
  node c (just (funs , l))
       (λ r → flip try (λ { (funs' , l') →
         hLam' E funs' m (st r l') }))
       (flip try λ { (funs' , lr) → hLam' E funs' m (k lr) })
```

The ‘abs case passes a closure value to k and (importantly!) *does not* apply the staged effect scope st to the latent effects yet. The ‘app case first unpacks the closure (via isClos), retrieves the function body in the resumption store, and then applies the function body to the *latent effects l for the application node*. The case for letbind illustrates how a scoped effect is a special case of a staged effect.

6 Discussion and Conclusion

The previous section has shown that with the Tree data type we can define operations in LambdaM, StateM, and NatM, and handle their effects in a modular way—we can add more operations without modifying their code. Below we discuss open questions about Tree.

Is Tree a free monad? The IO type of Hancock and Setzer [9] and the Prog type of Piróg et al. [17], Wu et al. [26] are free monads. We expect that the Tree type is too, by a similar line of reasoning as that of Piróg et al. [17, §4.1].

Recursion schemes. The IO type and the Prog types both admit notions of fold that factor out recursion. We expect that it is possible to define a similar notion of fold for Tree, and that this would work for defining the shown hSt” and a (scoped) handler for let bindings. However, hLam' has non-standard recursion, and would require reformulation to (possibly) fit into a fold based recursion scheme.

Staging beyond lambdas. In future work we will explore how to use staged effects and handlers to define the semantics of more interesting staging constructs, such as the staging abstractions found in MetaML [25] and related staging frameworks [2, 20, 22, 25].

Laws of LambdaM. Monadic operations are typically governed by laws that characterize their properties. These laws enable formal reasoning about the operations independent of their handler [8] and at the same time constrains handler implementations. As we plan to integrate our staged effects in the 3MT framework [6] in order to use LambdaM and other staging constructs in modular mechanized metatheory proofs, we will have to devise laws for them.

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*

342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

[2] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2830)*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer, 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

[3] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

[4] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, Richard A. DeMillo (Ed.). ACM Press, 207–212. https://doi.org/10.1145/582153.582176

[5] Laurence E. Day and Graham Hutton. 2013. Compilation à la Carte. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 28-30, 2013*, Rinus Plasmeijer (Ed.). ACM, 13. https://doi.org/10.1145/2620678.2620680

[6] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. 2013. Modular monadic meta-theory. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 319–330. https://doi.org/10.1145/2500365.2500587

[7] Martin Erwig and Tiark Rompf (Eds.). 2016. *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM. http://dl.acm.org/citation.cfm?id=2847538

[8] Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. *SIGPLAN Not.* 46, 9 (Sept. 2011), 2–14. https://doi.org/10.1145/2034574.2034777

[9] Peter G. Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1862)*, Peter Clote and Helmut Schwichtenberg (Eds.). Springer, 317–331. https://doi.org/10.1007/3-540-44622-2_21

[10] R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. http://www.jstor.org/stable/1995158

[11] Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Staging beyond terms: prospects and challenges, See [7], 103–108. https://doi.org/10.1145/2847538.2847548

[12] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

[13] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*, Jacques Carette and Jeremiah Willcock (Eds.). ACM, 13–24. https://doi.org/10.1145/2502488.2502491

[14] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

[15] Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2019. From definitional interpreter to symbolic executor. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, META@SPLASH 2019, Athens, Greece, October 20, 2019*, Christophe Scholliers and Guido Chari (Eds.). ACM, 11–20. https://doi.org/10.1145/3358502.3361269

[16] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[17] 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 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 809–818. https://doi.org/10.1145/3209108.3209166

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

[19] John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

[20] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[21] Dana Scott and Christopher Strachey. 1971. *Toward a Mathematical Semantics for Computer Languages*. Technical Report PRG-6. Oxford Programming Research Group.

[22] Tim Sheard and Simon L. Peyton Jones. 2002. Template metaprogramming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

[23] Jeremy G. Siek. 2013. Type Safety in Three Easy Lemmas. http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html.

[24] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[25] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*, John P. Gallagher, Charles Consel, and A. Michael Berman (Eds.). ACM, 203–217. https://doi.org/10.1145/258993.259019

[26] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 1–12. https://doi.org/10.1145/2633357.2633358

[27] Jeremy Yallop. 2016. Staging generic programming, See [7], 85–96. https://doi.org/10.1145/2847538.2847546