Effects with Variable Binding

Jaro Reinders
Delft University of Technology, The Netherlands
j.s.reinders@tudelft.nl

Casper Bach
University of Southern Denmark, Denmark
casperbach@imada.sdu.dk

Benedikt Ahrens
Delft University of Technology, The Netherlands
b.p.ahrens@tudelft.nl

Nicolas Wu
Imperial College London, United Kingdom
n.wu@imperial.ac.uk

Algebraic effects let programmers declare the syntax of operations which can be used to write and compose effectful programs. Such programs can be run by applying effect handlers that give programmers control over when and how many times to call continuations of operations. Embedding algebraic effects and handlers in an existing language provides a lightweight and powerful approach to programming with effects. Such embeddings typically use host language functions to represent continuations of programs. While host language functions offer an ergonomic interface for programming with and handling effects, they also hinder programmers wishing to define custom optimizations for effectful programs. Specifically, optimizations that statically analyze and transform variable bindings cannot be defined, as that would require meta-programming facilities for host language functions.

This paper presents a solution to this problem by embedding algebraic effects as intrinsically typed, De Bruijn-indexed syntax. Working in Agda, we demonstrate how this provides a safe-by-construction approach to programming with effects that is almost as ergonomic as the traditional embedding, but which additionally supports optimizations that inspect and transform variable binding.

1 Introduction

Algebraic effects provide a powerful abstraction for programming with effects. A main idea of the approach is to represent programs as a syntax tree of operations that can be manipulated by so-called *effect handlers* that evaluate the syntax tree [8, 31].

Such evaluation can, in principle, perform arbitrary analysis and manipulation of trees, such as interpreting operations, removing or duplicating them, or even statically analyzing the tree to facilitate optimization and enforce safety properties.

However, in practice, most stand-alone languages with algebraic effects [24, 25, 11], and most embeddings of algebraic effects and handlers in existing programming languages [37, 20, 41], focus on handlers that evaluate operations by, essentially, interpreting them, as we summarize in § 1.1. But for some applications, it is desirable to statically analyze and compile, rewrite, or otherwise transform syntax trees of operations. Existing approaches to algebraic effects and handlers do not readily support such analysis and transformation.

Submitted to:
MSFP 2026

1.1 Background: Algebraic Effects and Handlers

A simple example of an algebraic effect is an Output effect with a single operation:

$$\mathsf{out} : \mathsf{String} \to \mathsf{Free}\ \mathsf{Output}\ \top$$

Free σ A is the *free monad*, which is a syntax tree representing a computation whose return type is A, and whose effect signature is σ. Here σ is a *signature*, which may itself be a sum of smaller signatures, and which characterizes a set of operations.

Signatures such as Output declare a set of operations, which we can think of as an interface that programs can call. For Output, this interface consists of the single operation out, which accepts a string as argument and returns a unit value as result.

Effect handlers can be used to interpret the effects of a tree. For example, an effect handler for the Output effect could have the following type:

$$\mathsf{hOut} : \mathsf{Free}\ (\mathsf{Output} + \sigma)\ A \to \mathsf{Free}\ \sigma\ (A \times \mathsf{String})$$

This handler takes as input a computation that has return type A and that may call operations of both the Output effect and of the remaining signature σ. The handler returns as output a computation that may only call operations of σ and whose return type is now a product of the original return value type and a String which represents printed output.

The hOut handler illustrates a main selling point of algebraic effects and handlers: it interprets *only* the operations of the Output effect, leaving us free to apply different handlers to handle and interpret the remaining operations in σ.

Another key benefit of algebraic effects is that effects and handlers typically have associated equational theories that lets us use equational reasoning to verify properties of effectful programs.

In summary, algebraic effects and handlers provide a reusable, (typically) executable, and formal semantics of effects that supports verification.

1.2 Problem

Some applications require a semantics of effects that offers more fine-grained control over how programs will execute on the underlying hardware. Such applications require a different style of handler that transforms syntax trees of operations to optimize them, and to turn higher-level operations into lower-level ones. That is, rather than functions like hOut which *interpret* effects, we need functions that *compile* effects.

However, many common compiler transformations and optimizations require *binding aware* static analysis. As a simple example, say we have another effect with a simple operation for reading an input.

$$\mathsf{read} : \mathsf{Free}\ \mathsf{Input}\ \mathsf{String}$$

Consider a program that does not use the value that is produced by the read operation, in this case in this case just returning a unit value:

$$\mathsf{read} \gg\!= \lambda\ \_ \to \mathsf{return}\ \mathsf{tt}$$

Depending on the meaning we ascribe to the read operation, we could consider it dead code in this case because its result is discarded. So we might want to optimize it away:

```
return tt
```

The problem is that the traditional approach of representing algebraic effects and handlers models continuations (e.g., $\lambda \_ \to$ return tt) as *host language functions*. In most languages, such functions cannot be inspected or statically analyzed, which rules out the program optimization above, and many common compiler transformations like closure conversion and register allocation.

In summary, compiler writers could, in principle, use algebraic effects as an intermediate representation (IR) obtain similar benefits as discussed in § 1.1: reusable, executable, and formal semantics that supports verification via the equational theories associated with algebraic effects. However, the representation found in most existing approaches to algebraic effects and handlers precludes this.

1.3 Our Solution

We provide a solution to the problem of embedding algebraic effects in existing languages in a type safe way. Specifically, we present an approach to embedding algebraic effects in the dependently-typed programming language Agda [1]. Our approach should be straightforward to adapt to other dependently-typed languages (e.g., Coq, Lean, or Idris). Adapting to less dependently-typed languages (e.g., Haskell, Scala, Java, etc.) is also possible, but it may be harder to guarantee that programs are well-typed.

The main observation that motivates our solution, is that the problem with the traditional way of encoding algebraic effects and handlers in existing languages, is that they use higher-order abstract syntax (HOAS) [29, 14] to represent syntax trees of operations. That is, they use host language functions for binding. We remedy this problem by adapting the usual encoding of the free monad to use De Bruijn indices instead. We demonstrate how to realize this change in Agda and we show that it supports transformations representative of those found in compilers.

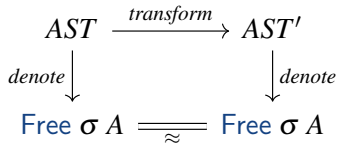
Since our approach is an (alternative) embedding of algebraic effects, it should be straightforward to extend our approach to support equational theories and equational reasoning, following, e.g., Yang and Wu [43] and Kidney et al. [21]. Doing so would, in principle, allow us to explore how to verify the correctness of compiler transformations. However, a necessary first step towards this exploration is the embedding we present and focus on in this work.

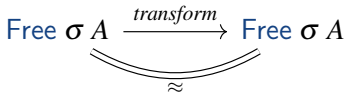
1.4 Relation to Existing Work

Previous work by Xia et al. [42] proposes to use algebraic effects for verifying compiler correctness, using an approach summarized by the following diagram on the left:[2]

[1] We have separately published a supporting code artifact containing the unedited Agda code [33].

[2] A technical difference from the work of Xia et al. is that they use a coinductive variant of the free monad, which they call *interaction trees*.





That is, they define a compiler using a classical AST-to-AST transformation, and verify its correctness, using the equational theory of the algebraic effects that each AST denotes to. However, this approach requires compiler writers to define and maintain separate denotation functions for each AST, including potentially separate semantics and notions of binding.

Instead, we propose a uniform approach for effects with variable binding, which lets us define compiler transformations directly on syntax trees of operations; see the diagram on the right above. The equality arrow represents a proof that *transform* is correct. Conducting such proofs using our framework is future work, but should follow a similar approach as the work by Xia et al.; namely, using the equational theory of the algebraic effects being transformed.

Our framework is also closely related to existing works on monadic IRs [38, 9]. A main difference from these works is that, like the work of [17], our framework uses the *free monad*, meaning it works for any set of operations. Another difference is that we embed our framework in Agda using safe-by-construction programming techniques. These safe-by-construction programming techniques are closely related to the work of Allais et al. [4] on *a type- and scope-safe universe of syntax with binding*. Our framework can be seen as an application and extension of their techniques to algebraic effects and monads.

1.5 Contributions

We proceed by illustrating what algebraic effects are and how they are traditionally embedded using HOAS, and why that is problematic (§ 2). Then, we make the following technical contributions:

- We present (§ 3) an intrinsically typed embedding of the free monad with variable binding in Agda.

- To demonstrate how our framework lets us analyze and transform programs, we present (§ 4) three case study transformations that are representative of the kinds of transformations that compilers do, and that are beyond what we can define using traditional embeddings of algebraic effects: (1) pretty-printing syntax trees defined using the free monad; (2) a compilation scheme that compiles programs with conditional expression operations to programs using low-level jump operations instead; and (3) a dead code elimination transformation that relies on a free variable analysis.

Finally, we position our framework relative to other work (§ 5), and then conclude (§ 6). We assume a basic familiarity with dependent types, category theory, and familiarity with Agda.

2 A Primer on Embedding Algebraic Effects in Agda

The traditional approach to working with algebraic effects and handlers uses host language functions for variable binding. In this section we show how to embed this traditional approach in Agda (or other dependently-typed languages), and the problem with this embedding.

The embedding we present is conceptually similar to embeddings of the free monad in, e.g., Haskell found in the literature [37, 41]. However, Agda's type system is stricter than Haskell's, which necessitates a slightly different encoding.

Hence, this section serves two purposes: (1) to provide background on the traditional approach to embed algebraic effects in Agda and its problems; and (2) to serve as a primer on encoding strictly-positive functors in Agda, as required for the free monad.

2.1 A Traditional Embedding in Agda, Using Data Type Descriptions

We can realize the free monad in Agda via the following data type:

```
data Free (σ : Desc) (A : Set) : Set where
  pure   : A                  → Free σ A
  impure : ⟦ σ ⟧ (Free σ A) → Free σ A
```

This data type has two constructors. The pure constructor represents a pure computation which returns a value of the expected return type. The impure constructor represents an operation generated by the signature functor $\llbracket\ \sigma\ \rrbracket$ (we will explain the type Desc and the function $\llbracket\_\rrbracket : \mathsf{Desc} \to \mathsf{Set} \to \mathsf{Set}$ shortly) whose continuation(s) are themselves computations of type Free $\sigma\ A$.

The type Desc are a form of *data type descriptions* [13] which in our case specify a subset of strictly-positive functors:[3]

```
data Desc : Set₁ where
  B   : List Set                → Desc
  K   : Set                     → Desc
  _⊡_ : Desc → Desc             → Desc
  ∐   : (X : Set) → (X → Desc) → Desc

⟦_⟧ : Desc → Set → Set
⟦ B Ts    ⟧ X = Tuple Ts → X
⟦ K T     ⟧ _ = T
⟦ σ₁ ⊡ σ₂ ⟧ X = ⟦ σ₁ ⟧ X × ⟦ σ₂ ⟧ X
⟦ ∐ S P   ⟧ X = Σ[ s ∈ S ] (⟦ P s ⟧ X)
```

The description B $(T_1 :: \cdots :: T_n :: [])$ represents a functor with $T_1 \ldots T_n$ parameter bindings; i.e., $F\,X = (T_1 \times \cdots \times T_n) \to X$. This semantics of descriptions is provided by the $\llbracket\_\rrbracket$ function.[4] Specifically, $\llbracket\ \mathsf{B}\ Ts\ \rrbracket\ A = \mathsf{Tuple}\ Ts \to A$, where:

$$\mathsf{Tuple}\ (T_1 :: \cdots :: T_n :: []) \cong T_1 \times \cdots \times T_n$$

The description K T represents a constant functor given by a value of type T. The description $\sigma_1 \dot{\times} \sigma_2$ represents a functor product of the functors given by the two descriptions σ_1 and σ_2,

[3] Alternatively one could use containers [1], but those are harder to adapt to a De Bruijn encoding as we will do later.

[4] Because each case of $\llbracket\_\rrbracket$ maps to a strictly-positive functor, Agda accepts its use in Free.

and the description $\coprod\ (s : S)\ P_s$ represents a functor coproduct with as many summands as S has inhabitants.

Example 1 (State signature functor) *The state monad functor can be defined as the following description:*[5]

```
data StateOp : Set where
  get-op : StateOp
  put-op : StateOp
```

```
State : Desc
State = ∐ StateOp λ where
  get-op → B (ℕ :: [])
  put-op → K ℕ ×̇ B []
```

The meaning of these descriptions can be computed as follows:

$$[\![\ B\ (\mathbb{N} :: [])\ ]\!]\ X\ =\ \mathit{Tuple}\ (\mathbb{N} :: [])\to X\ \simeq\ \mathbb{N}\to X$$
$$[\![\ K\ \mathbb{N}\ \dot{\times}\ B\ []\ ]\!]\ X\ =\ \mathbb{N}\times(\mathit{Tuple}\ []\to X)\ \simeq\ \mathbb{N}\times X$$

Using the State *description, we can define the following short-hands for operations whose signature matches the ones discussed in the introduction:*

```
get : Free State ℕ
get = impure (get-op , λ {(n :: []) → pure n})
```

```
put : ℕ → Free State ⊤
put n = impure (put-op , n , λ _ → pure tt)
```

Our Free embedding is a monad, with the following monadic bind operation in Agda:

```
_>>=_ : Free σ A → (A → Free σ B) → Free σ B
```

Its implementation is straightforward and elided for brevity here, but we return to it when we consider how to adapt the framework in § 3.

2.2 The Problem with the Traditional Embedding

As stated in the introduction, the goal of this paper is to bring the benefits of algebraic effects to compiler writers. As such, we desire a syntax tree of operations that we can inspect and statically analyze. Unfortunately, the traditional embedding described in § 2.1 does not readily support this.

Consider the following variation of the example we also discussed in the introduction, where $k : \mathbb{N} \to \mathsf{Free}\ \sigma\ A$ is some continuation:

$$\mathsf{get} \gg\!= k$$

Performing the dead code elimination optimization that gets rid of calls to get operations whose results are not used in k is a difficult task. If k is an arbitrary Agda function, it cannot be inspected

[5] The λ **where** . . . notation is a pattern matching lambda in Agda.

directly. The only way to inspect the structure of k is to apply it. However, since k dispatches on the natural numbers, it would need to be applied infinitely many times to be sure that all possible branches of the underlying function have been inspected.

This problem with functions in the representation of algebraic effects is not limited to the state effect. Any effectful operation that binds a variable (using B) suffers from the same problem. So, if we want to inspect and transform our effectful programs, we need a different representation for binding variables.

3 Algebraic Effects with Explicit Variable Binding

The previous section provided a primer on the usual approach to embedding algebraic effects in Agda. In this section, we present an alternative embedding. The main difference from § 2 is that we use a different underlying category.

We first briefly discuss our chosen category and its endofunctors in § 3.1, afterwards we present descriptions of functors and Free adapted to work on CSet in § 3.2, then we discuss the monad structure of this Free in § 3.3, and finally we discuss limitations in § 3.4.

3.1 The Category CSet

As summarized in § 2.2, the problem with the usual embedding of algebraic effects is that it uses host-language functions to bind intermediate results. Since most host languages—including Agda—do not allow intensional analysis of host language functions, applications that rely on such analysis—such as common compiler transformations—require a different embedding.

We propose to parameterize all types with contexts. Contexts are lists of types of all the variables that are in scope. In Agda we define this type (or, more accurately, kind) as follows:

```
CSet = List Set → Set₁
```

Theoretically, we intend this to be a functor category from renamings of the context to the category of set, a category theorist might write it as $[\mathbf{Ren}, \mathbf{Set}]$. This importantly means that we assume extra structure like the ability to rename variables in the context and associated functor laws, however for practical reasons we will only require those when necessary in Agda.

In CSet, we can define a data type for De Bruijn indices as follows:

```
data Var (A : Set) : CSet where
  here  : Var A (A :: Γ)
  there : Var A Γ → Var A (B :: Γ)
```

Furthermore, the category **CSet** is monoidally closed and has exponential objects:

```
_⇛_ : CSet → CSet → CSet
(P ⇛ Q) Γ = ∀{Γ′} → (∀{A} → Var A Γ → Var A Γ′) → P Γ′ → Q Γ′
```

3.2 Free on CSet

We define the carrier of the free monad on **CSet** as the following inductive data type in Agda:

```
data Free (σ : Desc) (P : CSet) : CSet where
  pure   : P Γ → Free σ P Γ
  impure : ⟦ σ ⟧ (Free σ P) Γ → Free σ P Γ
```

This definition resembles the definition of Free on Set from § 2. However, in order to accommodate the additional structure of CSet over Set, we will use a notion of description that distinguishes *variables* from *constants*. The semantics of these descriptions are endofunctors on CSet:

```
data Desc : Set₁ where
  B   : List Set → Desc
  V   : Set → Desc
  K   : Set → Desc
  _⊡_ : Desc → Desc → Desc
  ∐   : (X : Set) → (X → Desc) → Desc
```

```
⟦_⟧ : Desc → CSet → CSet
⟦ B Ts    ⟧ P Γ = P (Ts ++ Γ)
⟦ V T     ⟧ P Γ = Var T Γ
⟦ K T     ⟧ _ _ = Lift _ T
⟦ σ₁ ⊡ σ₂ ⟧ P Γ = ⟦ σ₁ ⟧ P Γ × ⟦ σ₂ ⟧ P Γ
⟦ ∐ S P   ⟧ Q Γ = Σ[ s ∈ S ] (⟦ P s ⟧ Q Γ)
```

The key differences from in § 2.1 are in the first three cases. The first case (B) now extends the context. The second case (V) represents a *variable reference*. The third case (K) represents a *constant*; i.e., an Agda value that is independent of the context. The fourth and fifth cases represent products and coproducts, as in § 2.1, except in CSet.

Example 2 *Using the definitions above, we can define signature functors similar to the ones discussed in the introduction and in § 2.1. For example, the state functor:*

```
data StateOp : Set where
  get-op : StateOp
  put-op : StateOp
```

```
State : Desc
State = ∐ StateOp λ where
  get-op → B (ℕ :: [])
  put-op → V ℕ ⊡ B []
```

Using this, we can write programs using Free. The following program gets the state and returns it:

```
get : Free State (Var ℕ) Γ
get = impure (get-op , pure here)
```

The initial algebra given by Free is also associated with a notion of catamorphism (or fold) which lets us interpret syntax trees.

```
fold : (P ⇛ Q) Γ → (⟦ σ ⟧ Q ⇛ Q) Γ → Free σ P Γ → Q Γ
fold          gen alg (pure x)   = gen id x
fold {σ = σ} gen alg (impure x) = alg id (map-fold σ gen alg x)
```

Morally, the call to *alg* in the impure case maps a recursive call to the fold over the functor $[\![\ \sigma\ ]\!]$. However, a direct recursive call is not accepted by Agda's termination checker. To appease the termination checker, we therefore use a specialized helper function map-fold that achieves the same effect. This standard specialized helper trick [27, §3] is elided for brevity.

The syntax tree of operations and its associated catamorphism lets us inspect and manipulate the binding structure of the free monad. To illustrate, let us revisit the problematic example from the introduction and § 2.2; i.e., a program get $\gg\!=$ k where we want to know whether the continuation k may use its parameter. Our CSet-based free monad lets us count the number of occurrences of each variable in the context of a given computation.

In § 4 we provide examples that demonstrate how we can also transform terms based on such analysis. First, we provide a monadic bind and generic fold for the free monad with variable binding.

3.3 The Monad Structure of Free

We study the monadic structure of Free through the lens of *Kleisli triples*, which involves a Kleisli extension function and two laws which it has to satisfy. The Kleisli extension function is defined using the following signature:

```
extend : (∀{Γ} → P Γ → Free σ Q Γ) → Free σ P Γ → Free σ Q Γ
```

This can be implemented for our Free data type using a simple fold:

```
extend f = fold (λ _ → f) (λ _ → impure)
```

However, this standard formulation of the extension function is problematic which becomes apparent if you try using it. Say we want to encode the program written as follows using **do** notation:

```
do x ← get; y ← get; pure x
```

Replacing the monadic bind in this program by the Kleisli extension on CSet, the following program does not type check! The highlighted x below is ill-typed:

```
extend (λ x → extend (λ y → pure x) get) get
```

The problem is that the x bound by the outermost lambda is indexed by a *different* context than the inner lambda which binds y.

To remedy this, we need to generalize our notion of monad to have Kleisli extension function which is a morphisms between exponential objects in **CSet**:

```
extend : (P ⇛ Free σ Q) Γ → (Free σ P ⇛ Free σ Q) Γ
extend k r₁ m = fold (λ r₂ → k (r₂ ∘ r₁)) (λ _ → impure) m
```

Flipping the arguments of this generalized Kleisli extension gives us the monadic bind familiar to Haskell programmers.

```
_>>=_ : Free σ P Γ → (P ⇛ Free σ Q) Γ → Free σ Q Γ
m >>= k = extend k id m
```

Using this, the program from before now type checks, by applying the appropriate renaming:

```
prog : Free State (Var ℕ) Γ
prog = get >>= λ _ x → get >>= λ r y → pure (r x)
```

A full definition of this generalized monad and a proof that Free is a lawful instance can be found in our supporting code artifact [33].

3.4 Discussion and Limitations

We have shown that the free monad with variable binding lets us analyze effectful programs, and that it enjoys the compositionality that we have come to expect from free monads. However, there are two limitations that crop up when applying this representation in practice.

The first limitation is that bound variables in our representation are no longer variables in the host language, so we cannot easily use Agda functions to manipulate the variables in our representation (§ 3.4.1). The second limitation is that the intermediate representation may be much larger than necessary if it involves many branching operations (§ 3.4.2).

3.4.1 Limitation I: Cannot Use Host Language Functions

We have identified that the use of host language functions in our representation is the fundamental problem that prevents analysis and transformation. Eschewing host language functions, however, means that we have to essentially define the whole language within our framework. This is the price we pay for full control.

Consider, for example, the simple program **do** put 1; get. Using the free monad with variable binding, we cannot directly use the constant 1 like we did in the previous section, because such a constant is not a Var, which is what the put operation expects. We could write the following program instead:

```
state-ex₀ : Var ℕ Γ → Free State (Var ℕ) Γ
state-ex₀ x = put x >>= λ _ _ → get
```

But that still leaves the question of how to produce the value Var ℕ.

In order to use number constants in programs, we can declare an effect that gives us that ability with operations for embedding an Agda natural number into our DSL and a control operation which enables matching on those natural numbers.

Using such a Nat description we can write a more faithful version of the example program:

```
state-ex₁ : Free (State + Nat) (Var ℕ) Γ
state-ex₁ = intro 1 >>= λ _ one → put one >>= λ _ _ → get
```

This seems convoluted if you only consider constants, but you face the same issue for many other language constructs, like arithmetic or pattern matching. Furthermore, in a compiler pipeline you want to support different language constructs at different stages of compilation: the source language might support pattern matching, but a low-level assembly language won't. We could alternatively interpret the variables in our descriptions as more than just De Bruijn indices, but supporting all desired language constructs would require a complicated extensible expression type à la carte [37]. Thus, we opted for the simplicity of just adding extra effect operations as described above.

3.4.2 Limitation II: monadic bind can cause exponential representation size

The monadic bind operation that we have introduced has one important flaw which makes it impractical to use for branching operations such as the match operation. The problem is that the bind operation will propagate the continuation it is given into all branches of all operations it encounters. That way, a program written using multiple binds and multiple branching operations can yield an intermediate representation which grows exponentially in size.

This is not an issue when lazily running an interpreter over the free monad, but it is problematic if you want to write program transformations on this representation. We are considering several solutions to this problem like an explicit bind constructor with delayed propagation, or introducing a `let`-like construct for explicit sharing of continuations. However, we leave the full development of these ideas to future work.

4 Case studies

We have presented our representation of effectful programs as a free monad with explicit variables. In the process, we have shown simple examples of programs and program analyses, but we have not yet shown program transformations.

In this section, we show three representative case studies of program transformations. Our first case study is that of pretty-printing our programs, which shows that our representation is fully concrete (§ 4.1). Our second case study is a desugaring pass of conditionals to jumps, which shows that our representation can model sophisticated effects like sub/jump and that simple desugaring transformations are still simple on our representation (§ 4.2). Finally, our third case study is that of free variable analysis and dead code elimination, which shows that our representation is capable of advanced context manipulations (§ 4.3).

We have elided many implementation details in this section, we refer the interested reader to our supporting code artifact which contains the unedited implementations [33].

4.1 Pretty-Printing

Pretty-printing is a task that a HOAS representation would struggle to perform. We demonstrate it here to show that our representation is fully concrete.

We use a record Output to define what it means to convert operations to strings. This will be passed around implicitly like a type class, and thus allow the user to tailor the output of each effect. If a user can define what it means to output their effects individually then this pretty-printing code can pretty-print any computation that uses those effects. In this way, our pretty-printing transformation inherits the modularity of algebraic effects.[6]

```
record Output (σ : Desc) : Set₂ where
  field output : (∀{T} → Var T Γ → String) → ⟦ σ ⟧ P Γ → String
```

The pretty-printing transformation takes care of all the structure around each operation, such as the binding of variables, branching, and opening and closing do-blocks. For that purpose, it uses the structure of the free monad and effect signature.

```
pretty : {{ _ : Output σ }} → Free σ (Tuple Ts) [] → String
```

4.2 Compiling Conditionals to Jumps

Our second case study compiles conditionals to blocks and jumps. This case study illustrates that our representation can handle more sophisticated effects. The transformation itself is simple and could probably be implemented as a handler in other algebraic effect systems. However, we show that such transformations are still simple to implement in our representation.

The input effect of the transformation is Cond, which has one operation: if-then-else. The if-then-else operation represents a conditional, and takes a boolean variable as input and two continuations which bind no variables. If the boolean is true then it takes the first continuation, otherwise it takes the second continuation.

```
Cond : Desc
Cond = V Bool ×̇ (B [] ×̇ B [])
```

The transformation translates this Cond effect into an effect with blocks and conditional jumps.

The Block effect has two operations: introducing a new block with block, and conditional branching brIf. The block operation has two continuations. The first continuation binds a reference to the second continuation. The brIf operation can perform the actual jump to the other continuation based on the value of a boolean variable.

```
data BlockOp : Set where
  block-op : BlockOp
  brIf-op  : BlockOp
```

```
Block : (Ref : Set) → Desc
Block Ref = ∐ BlockOp λ where
  block-op → B (Ref :: []) ×̇ B []
  brIf-op  → (V Ref ×̇ V Bool) ×̇ B []
```

[6] The definition of StateOutput uses Agda *copattern matching*: `https://agda.readthedocs.io/en/v2.6.4.3-r1/language/copatterns.html`

```
block : {{ _ : Block Ref < σ }} → (Var Ref ⇛ Free σ P) Γ → Free σ P Γ → Free σ P Γ
brIf  : {{ _ : Block Ref < σ }} → Var Bool Γ → Var Ref Γ → Free σ (Tuple []) Γ
```

The transformation itself looks for occurrences of the if-then-else operation and replaces it by a combination of block and brIf. This function can be applied to any effectful computation that includes the Cond effect.

```
compile : ∀ {{ _ : Mono P }} {{ _ : Block Ref < σ }}
        → Free (Cond + σ) (Tuple Ts) Γ → Free σ (Tuple Ts) Γ
compile = fold (λ _ → pure) (λ _ → λ where
  (true , b , t , f) → block (λ e ref → brIf (e b) ref >>= λ e' _ → wk (e' ∘ e) f) t
  (false , x) → impure x)
```

4.3 Dead Code Elimination

The final case study shows how we can remove redundant operations from effectful computations. This finally makes true on the promise we made in the introduction. The goal of dead code elimination is to remove operations whose result is unused and who are not otherwise causing observable effects. Our principal example is a computation that uses the get operation:

$$\textbf{do}\ x \leftarrow \mathsf{get};\ k\ x$$

If k does not use its formal parameter, then the get operation is redundant and can be removed without changing the meaning of this computation.

In contrast, it is not sound to eliminate the put operation, since it has an observable side-effect. In order to define a dead code elimination transformation that works for arbitrary operations, we need a characterization of which operations have observable side-effects (such as put) and which do not (such as get). Our transformation will therefore require users to provide instances of the following record type for each signature, to specify which operations are observable.

```
record IsObservable (σ : Desc) : Set₂ where
  field isObservable : ⟦ σ ⟧ P Γ → Bool
```

We usually do not want to be explicit about what the actual "thinned" [28] context is, so we introduce a type that existentially quantifies over the thinned context:

```
◇ : CSet → CSet
◇ P Γ = Σ[ Γ' ∈ List Set ] Γ' ⊆ Γ × P Γ'
```

Note that we are usually interested in the smallest Γ' such that $P\ \Gamma'$ is still well typed, but we do not make that explicit in our types.[7]

[7] One could use co-de-Bruijn [28] for minimal contexts by construction, but we chose to keep it simple.

This diamond type has some nice properties, such as it being an endofunctor on CSet (this time we do not need the enriched structure): Additionally, we can combine two elements of this diamond type.

The dead code elimination function itself is a fold:

```
elimDeadCode : {{ _ : IsObservable σ }} → Free σ (Tuple Ts) Γ → ◇ (Free σ (Tuple Ts)) Γ
```

To illustrate that this lets us eliminate dead code, we apply dead code elimination to a simple example program which containing a redundant get operation:

```
prog′ : Free State (Tuple (ℕ :: [])) []
prog′ = get >>= λ _ x → get >>= λ e _ → pure (wk e x)

test : (case elimDeadCode {{ IsObservableState }} prog′ of λ where
          ([] , _ , x) → Pretty.pretty x)
       ≡ "do { $0 <- get; pure ($0 :: []) }"
test = refl
```

5 Related work

Algebraic effects and handlers are emerging as a popular programming paradigm for programming with effects. This paradigm is being actively explored and developed on in both libraries [20, 30, 39] and in standalone languages [24, 11, 36]. One of the main benefits of the paradigm is its modularity [44]: programs can easily be extended with new effects, and the semantics of effects can easily be refined. One reason why it is attractive to be able to refine and give multiple semantics of handlers is that it is useful to have both a reference semantics and an efficiently implemented semantics of effects. One way to obtain a more efficient semantics of effectful operations is to define optimizations akin to those found in standard compilers, and translating effectful programs into lower-level, more efficiently executable operations.

Existing works mainly focus on general-purpose techniques for compiling arbitrary effectful programs, such as effect handler fusion [40], CPS transformations [24, 18], parallelization [36], and better memory management [34]. In contrast, the framework we have presented enables library developers and language designers to define their own custom optimizations for effectful programs that can exploit domain-specific knowledge about particular effects and application domains. Enabling language designers to define their own custom optimizations on a modular IR is also the idea behind MLIR [23]. While MLIR does not have a formal semantics proper, recent work by Bhat et al. [10] explores a mechanization of MLIR and peephole optimizations. An interesting question for future work is to use the free monad with variable binding (§ 3) to provide an executable, formal, and modular semantics for a compiler IR which supports formally verified optimizations.

To support this use case, it will be desirable to also support *higher-order operations*; i.e., operations that may have computational parameters that do not represent continuations, such as an exception catching operation $\mathsf{catch} : \mathsf{Free}\ \sigma\ A \to (\mathit{Exc} \to \mathsf{Free}\ \sigma\ A) \to \mathsf{Free}\ \sigma\ A$. Previous work by Poulsen and van der Rest [32]; van den Berg and Schrijvers [39]; Yang and Wu [44] demonstrate how the *higher-order free monad* supports such operations. We expect that the lifting from Set to CSet should be equally possible for the higher-order free monad as for the traditional free monad considered in this paper.

One angle of our work is to provide an approach to *unembedding* [26, 6, 7] of the traditional embedding of algebraic effects we discussed in § 2. That is, we convert from HOAS to De Bruijn indexed terms. Unembedding [6, 7] provides a proof of correspondence between HOAS and De Bruijn indexed terms. Instead of providing this proof of correspondence, we show how algebraic effects—specifically, the free monad and algebras over it—can be embedded as De Bruijn indexed terms in Agda, exploiting the categorical foundations of syntax with variable binding. Our resulting embedding lets us program with *smart constructors* and *monadic bind*, akin to their plain HOAS counterparts.

Algebraic theories of syntax with variable binding have been devised since the late 1990's. In particular, Hofmann [19], Gabbay and Pitts [16], and Fiore, Plotkin, and Turi [15] studied variable binding in the form of HOAS, nominal syntax, and De Bruijn syntax, respectively. Fiore et al.'s approach was later reformulated in terms of relative monads, see, e.g., work by Ahrens [2] and Altenkirch et al. [5]. An extensive overview of algebraic characterizations of abstract syntax forms of De Bruijn variables is given by Lamiaux and Ahrens [22].

The formalization of syntax in computer proof assistants was developed in many different lines of work. In the context of initial algebra semantics, Ahrens [2] and Ahrens and Zsidó [3] provide a notion of signature with, for each signature, its associated type-safe and scope-safe syntax and generic folding operations, in particular, substitution. In the context of functional programming, similar work has been done by Allais et al. [4] and Chapman et al. [13]. In particular, a notion of data type descriptions, as a specific form of signature, was introduced by Chapman et al. [13].

6 Conclusion

We have presented a framework for programming with effects with variables. This framework lets us perform binding-aware analysis, transformation, and optimization of effectful programs, beyond what traditional HOAS-based embeddings of algebraic effects support. While our framework uses explicit De Bruijn indices, we have shown it is based on the same categorical definitions (i.e., the free monad and algebras over it) as traditional embeddings.

We have demonstrated the expressiveness that the free monad with variable binding affords on three case studies: a pretty-printer (§ 4.1) demonstrating how we can intensionally inspect syntax; a code generation case study (§ 4.2) that compiles higher-level conditional operations into lower-level blocks and jumps, demonstrating that implementing transformations is simple; and a static analysis and code optimization case study (§ 4.3) which shows that we support the kinds of

optimizations that traditional HOAS-based embeddings of algebraic effects do not.

Our work enables several future research directions at the intersection of the research areas of algebraic effects and compilers. In particular, we propose to explore how to use the constructions presented in this paper for compiler IRs that (1) are well-typed by construction, (2) have an executable and compositional semantics, and (3) have equational theories that could be used to reason about and verify compiler transformations. To explore this, the constructions presented in this paper should be adapted to a higher-order effects setting, and should be extended with laws and equational reasoning, ideally in a modular fashion explored in the works of Schrijvers et al. [35], Yang and Wu [43], and Kidney et al. [21].

References

[1] Michael Abbott, Thorsten Altenkirch & Neil Ghani (2003): *Categories of Containers*. In Andrew D. Gordon, editor: *Foundations of Software Science and Computation Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 23–38.

[2] Benedikt Ahrens (2016): *Modules over relative monads for syntax and semantics*. *Math. Struct. Comput. Sci.* 26(1), pp. 3–37, doi:10.1017/S0960129514000103. Available at `https://doi.org/10.1017/S0960129514000103`.

[3] Benedikt Ahrens & Julianna Zsido (2011): *Initial Semantics for higher-order typed syntax in Coq*. *J. Formaliz. Reason.* 4(1), pp. 25–69, doi:10.6092/ISSN.1972-5787/2066. Available at `https://doi.org/10.6092/issn.1972-5787/2066`.

[4] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride & James McKinna (2018): *A type and scope safe universe of syntaxes with binding: their semantics and proofs*. *Proc. ACM Program. Lang.* 2(ICFP), pp. 90:1–90:30, doi:10.1145/3236785. Available at `https://doi.org/10.1145/3236785`.

[5] Thorsten Altenkirch, James Chapman & Tarmo Uustalu (2015): *Monads need not be endofunctors*. *Log. Methods Comput. Sci.* 11(1), doi:10.2168/LMCS-11(1:3)2015. Available at `https://doi.org/10.2168/LMCS-11(1:3)2015`.

[6] Robert Atkey (2009): *Syntax for Free: Representing Syntax with Binding Using Parametricity*. In Pierre-Louis Curien, editor: *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, *Lecture Notes in Computer Science* 5608, Springer, pp. 35–49, doi:10.1007/978-3-642-02273-9_5. Available at `https://doi.org/10.1007/978-3-642-02273-9_5`.

[7] Robert Atkey, Sam Lindley & Jeremy Yallop (2009): *Unembedding domain-specific languages*. In Stephanie Weirich, editor: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, ACM, pp. 37–48, doi:10.1145/1596638.1596644. Available at `https://doi.org/10.1145/1596638.1596644`.

[8] Andrej Bauer & Matija Pretnar (2015): *Programming with algebraic effects and handlers*. *J. Log. Algebraic Methods Program.* 84(1), pp. 108–123, doi:10.1016/j.jlamp.2014.02.001. Available at `https://doi.org/10.1016/j.jlamp.2014.02.001`.

[9] Nick Benton, Andrew Kennedy & George Russell (1998): *Compiling Standard ML to Java Bytecodes*. In Matthias Felleisen, Paul Hudak & Christian Queinnec, editors: *Proceedings of the third*

ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998, ACM, pp. 129–140, doi:10.1145/289423.289435. Available at https://doi.org/10.1145/289423.289435.

[10] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens & Tobias Grosser (2024): *Verifying Peephole Rewriting In SSA Compiler IRs*. In: *The International Conference on Interactive Theorem Proving*, *23* ITP 2024, pp. 1–20.

[11] Jonathan Immanuel Brachthäuser, Philipp Schuster & Klaus Ostermann (2020): *Effects as capabilities: effect handlers and lightweight effect polymorphism*. *Proc. ACM Program. Lang.* 4(OOPSLA), pp. 126:1–126:30, doi:10.1145/3428194. Available at https://doi.org/10.1145/3428194.

[12] Giuseppe Castagna & Andrew D. Gordon, editors (2017): *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, doi:10.1145/3009837. Available at https://doi.org/10.1145/3009837.

[13] James Chapman, Pierre-Évariste Dagand, Conor McBride & Peter Morris (2010): *The gentle art of levitation*. In Paul Hudak & Stephanie Weirich, editors: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, ACM, pp. 3–14, doi:10.1145/1863543.1863547. Available at https://doi.org/10.1145/1863543.1863547.

[14] Joëlle Despeyroux, Amy P. Felty & André Hirschowitz (1995): *Higher-Order Abstract Syntax in Coq*. In Mariangiola Dezani-Ciancaglini & Gordon D. Plotkin, editors: *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, *Lecture Notes in Computer Science* 902, Springer, pp. 124–138, doi:10.1007/BFB0014049. Available at https://doi.org/10.1007/BFb0014049.

[15] Marcelo P. Fiore, Gordon D. Plotkin & Daniele Turi (1999): *Abstract Syntax and Variable Binding*. In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, IEEE Computer Society, pp. 193–202, doi:10.1109/LICS.1999.782615. Available at https://doi.org/10.1109/LICS.1999.782615.

[16] Murdoch Gabbay & Andrew M. Pitts (1999): *A New Approach to Abstract Syntax Involving Binders*. In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, IEEE Computer Society, pp. 214–224, doi:10.1109/LICS.1999.782617. Available at https://doi.org/10.1109/LICS.1999.782617.

[17] Neil Ghani, Tarmo Uustalu & Makoto Hamana (2006): *Explicit substitutions and higher-order syntax*. *High. Order Symb. Comput.* 19(2-3), pp. 263–282, doi:10.1007/S10990-006-8748-4. Available at https://doi.org/10.1007/s10990-006-8748-4.

[18] Daniel Hillerström, Sam Lindley & Robert Atkey (2020): *Effect handlers via generalised continuations*. *J. Funct. Program.* 30, p. e5, doi:10.1017/S0956796820000040. Available at https://doi.org/10.1017/S0956796820000040.

[19] Martin Hofmann (1999): *Semantical Analysis of Higher-Order Abstract Syntax*. In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, IEEE Computer Society, pp. 204–213, doi:10.1109/LICS.1999.782616. Available at https://doi.org/10.1109/LICS.1999.782616.

[20] Ohad Kammar, Sam Lindley & Nicolas Oury (2013): *Handlers in action*. In Greg Morrisett & Tarmo Uustalu, editors: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13,*

Boston, MA, USA - September 25 - 27, 2013, ACM, pp. 145–158, doi:10.1145/2500365.2500590. Available at https://doi.org/10.1145/2500365.2500590.

[21] Donnacha Oisín Kidney, Zhixuan Yang & Nicolas Wu (2024): *Algebraic Effects Meet Hoare Logic in Cubical Agda*. *Proc. ACM Program. Lang.* 8(POPL), doi:10.1145/3632898. Available at https://doi.org/10.1145/3632898.

[22] Thomas Lamiaux & Benedikt Ahrens (2024): *An Introduction to Different Approaches to Initial Semantics*. *CoRR* abs/2401.09366, doi:10.48550/ARXIV.2401.09366. arXiv:2401.09366.

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache & Oleksandr Zinenko (2021): *MLIR: scaling compiler infrastructure for domain specific computation*. In: *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, IEEE Press, p. 2–14, doi:10.1109/CGO51591.2021.9370308. Available at https://doi.org/10.1109/CGO51591.2021.9370308.

[24] Daan Leijen (2017): *Type directed compilation of row-typed algebraic effects*. In Castagna & Gordon [12], pp. 486–499, doi:10.1145/3009837.3009872. Available at https://doi.org/10.1145/3009837.3009872.

[25] Sam Lindley, Conor McBride & Craig McLaughlin (2017): *Do be do be do*. In Castagna & Gordon [12], pp. 500–514, doi:10.1145/3009837.3009897. Available at https://doi.org/10.1145/3009837.3009897.

[26] Kazutaka Matsuda, Samantha Frohlich, Meng Wang & Nicolas Wu (2023): *Embedding by Unembedding*. *Proc. ACM Program. Lang.* 7(ICFP), pp. 1–47, doi:10.1145/3607830. Available at https://doi.org/10.1145/3607830.

[27] Conor McBride (2011): *Ornamental Algebras, Algebraic Ornaments*. Unpublished manuscript.

[28] Conor McBride (2018): *Everybody's Got To Be Somewhere*. *Electronic Proceedings in Theoretical Computer Science* 275, p. 53–69, doi:10.4204/eptcs.275.6. Available at http://dx.doi.org/10.4204/EPTCS.275.6.

[29] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In Richard L. Wexelblat, editor: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, ACM, pp. 199–208, doi:10.1145/53990.54010. Available at https://doi.org/10.1145/53990.54010.

[30] Maciej Piróg, Tom Schrijvers, Nicolas Wu & Mauro Jaskelioff (2018): *Syntax and Semantics for Operations with Scopes*. In Anuj Dawar & Erich Grädel, editors: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, ACM, pp. 809–818, doi:10.1145/3209108.3209166. Available at https://doi.org/10.1145/3209108.3209166.

[31] Gordon D. Plotkin & Matija Pretnar (2009): *Handlers of Algebraic Effects*. In Giuseppe Castagna, editor: *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* 5502, Springer, pp. 80–94, doi:10.1007/978-3-642-00590-9_7. Available at https://doi.org/10.1007/978-3-642-00590-9_7.

[32] Casper Bach Poulsen & Cas van der Rest (2023): *Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects*. *Proc. ACM Program. Lang.* 7(POPL), pp. 1801–1831, doi:10.1145/3571255. Available at https://doi.org/10.1145/3571255.

[33] Jaro Reinders (2026): *Supporting code for Effects with Variable Binding*, doi:10.5281/zenodo.20071761. Available at https://doi.org/10.5281/zenodo.20071761.

[34] Alex Reinking, Ningning Xie, Leonardo de Moura & Daan Leijen (2021): *Perceus: garbage free reference counting with reuse*. In Stephen N. Freund & Eran Yahav, editors: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, ACM, pp. 96–111, doi:10.1145/3453483.3454032. Available at https://doi.org/10.1145/3453483.3454032.

[35] Tom Schrijvers, Maciej Piróg, Nicolas Wu & Mauro Jaskelioff (2019): *Monad transformers and modular algebraic effects: what binds them together*. In Richard A. Eisenberg, editor: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, ACM, pp. 98–113, doi:10.1145/3331545.3342595. Available at https://doi.org/10.1145/3331545.3342595.

[36] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer & Anil Madhavapeddy (2021): *Retrofitting effect handlers onto OCaml*. In Stephen N. Freund & Eran Yahav, editors: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, ACM, pp. 206–221, doi:10.1145/3453483.3454039. Available at https://doi.org/10.1145/3453483.3454039.

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

[38] Andrew P. Tolmach (1998): *Optimizing ML Using a Hierarchy of Monadic Types*. In: *Proceedings of the Second International Workshop on Types in Compilation*, TIC '98, Springer-Verlag, Berlin, Heidelberg, p. 97–115.

[39] Birthe van den Berg & Tom Schrijvers (2024): *A framework for higher-order effects & handlers*. *Science of Computer Programming* 234, p. 103086, doi:https://doi.org/10.1016/j.scico.2024.103086. Available at https://www.sciencedirect.com/science/article/pii/S0167642324000091.

[40] Nicolas Wu & Tom Schrijvers (2015): *Fusion for Free*. In Ralf Hinze & Janis Voigtländer, editors: *Mathematics of Program Construction*, Springer International Publishing, Cham, pp. 302–322.

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

[42] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce & Steve Zdancewic (2020): *Interaction trees: representing recursive and impure programs in Coq*. *Proc. ACM Program. Lang.* 4(POPL), pp. 51:1–51:32, doi:10.1145/3371119. Available at https://doi.org/10.1145/3371119.

[43] Zhixuan Yang & Nicolas Wu (2021): *Reasoning about effect interaction by fusion*. *Proc. ACM Program. Lang.* 5(ICFP), doi:10.1145/3473578. Available at https://doi.org/10.1145/3473578.

[44] Zhixuan Yang & Nicolas Wu (2023): *Modular Models of Monoids with Operations*. *Proc. ACM Program. Lang.* 7(ICFP), doi:10.1145/3607850. Available at https://doi.org/10.1145/3607850.