Closure-Free Functional Programming in a Two-Level Type Theory

ANONYMOUS AUTHOR(S)

Many abstraction tools in functional programming rely heavily on general-purpose compiler optimization to achieve adequate performance. For example, monadic binding is a higher-order function which yields runtime closures in the absence of sufficient compile-time inlining and beta-reductions, thereby significantly degrading performance. In current systems such as the Glasgow Haskell Compiler, there is no strong guarantee that general-purpose optimization can eliminate abstraction overheads, and users only have indirect and fragile control over code generation through inlining directives and compiler options. We propose a two-stage language to simultaneously get strong guarantees about code generation and strong abstraction features. The object language is a simply-typed first-order language which can be compiled without runtime closures. The compile-time language is a dependent type theory. The two are integrated in a two-level type theory.

We demonstrate two applications of the system. First, we develop monads and monad transformers. Here, abstraction overheads are eliminated by staging and we can reuse almost all definitions from the existing Haskell ecosystem. Second, we develop pull-based stream fusion. Here we make essential use of dependent types to give a concise definition of a concatMap operation with guaranteed fusion. We provide an Agda implementation and a typed Template Haskell implementation of these developments.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Source code generation**.

Additional Key Words and Phrases: two-level type theory, staged compilation

ACM Reference Format:
Anonymous Author(s). 2024. Closure-Free Functional Programming in a Two-Level Type Theory. 1, 1 (February 2024), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern functional programming supports many convenient abstractions. These often come with significant runtime overheads. Sometimes the overheads are acceptable, but in other cases compiler optimization is crucial. Monads in Haskell is an example for the latter. Even the Reader monad, which is one of the simplest in terms of implementation, yields large overheads when compiled without optimizations. Consider the following:

```
f :: Int → Reader Bool Int
f x = do {b ← ask; if x then return (x + 10) else return (x + 20)}
```

With optimizations enabled, GHC compiles this roughly to the code below:

```
f :: Int → Bool → Int
f = λ x b. if b then x + 10 else x + 20
```


ACM XXXX-XXXX/2024/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Closure-Free Functional Programming in a Two-Level Type Theory

FIRST LAST, Institution, Country

Many abstraction tools in functional programming rely heavily on general-purpose compiler optimization to achieve adequate performance. For example, monadic binding is a higher-order function which yields runtime closures in the absence of sufficient compile-time inlining and beta-reductions, thereby significantly degrading performance. In current systems such as the Glasgow Haskell Compiler, there is no strong guarantee that general-purpose optimization can eliminate abstraction overheads, and users only have indirect and fragile control over code generation through inlining directives and compiler options. We propose a two-stage language to simultaneously get strong guarantees about code generation and strong abstraction features. The object language is a simply-typed first-order language which can be compiled without runtime closures. The compile-time language is a dependent type theory. The two are integrated in a two-level type theory.

We demonstrate two applications of the system. First, we develop monads and monad transformers. Here, abstraction overheads are eliminated by staging and we can reuse almost all definitions from the existing Haskell ecosystem. Second, we develop pull-based stream fusion. Here we make essential use of dependent types to give a concise definition of a concatMap operation with guaranteed fusion. We provide an Agda implementation and a typed Template Haskell implementation of these developments.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Source code generation**.

Additional Key Words and Phrases: two-level type theory, staged compilation

ACM Reference Format:
First Last. 2024. Closure-Free Functional Programming in a Two-Level Type Theory. 1, 1 (June 2024), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern functional programming supports many convenient abstractions. These often come with significant runtime overheads. Sometimes the overheads are acceptable, but in other cases compiler optimization is crucial. Monads in Haskell is an example for the latter. Even the Reader monad, which is one of the simplest in terms of implementation, yields large overheads when compiled without optimizations. Consider the following:

```
f :: Int → Reader Bool Int
f x = do {b ← ask; if x then return (x + 10) else return (x + 20)}
```

With optimizations enabled, GHC compiles this roughly to the code below:

```
f :: Int → Bool → Int
f = λ x b. if b then x + 10 else x + 20
```

Author's address: First Last, Institution, Country, City.

ACM XXXX-XXXX/2024/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Without optimizations we roughly get:

```
f = λ x. (>>=) MonadReaderDict ask (λ b. if b
  then return MonadReaderDict (x + 10)
  else return MonadReaderDict (x + 20))
```

Here, MonadReaderDict is a runtime dictionary, containing the methods of the Monad instance for Reader, and (>>=) MonadReaderDict is a field projection. Here, a runtime closure will be created for the λ b. ... function, and (>>=), ask and **return** will create additional dynamic closures.

The difference between optimized and unoptimized code is already large here, and it gets even larger when we consider monad transformers or code that is polymorphic over monads. In Haskell, such code is pervasive, even in fairly basic programs which do not use fancy abstractions. Consider the mapM function from the Haskell Prelude:

```
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
```

This is a third-order and second-rank polymorphic function in disguise, because its monad dictionary argument contains the polymorphic second-order method (>>=). Compiling mapM efficiently relies on inlining the instance dictionary, then inlining the methods contained there, and also inlining the functions that the higher-order binding method is applied to.

GHC's optimization efforts are respectable, and it has gotten quite adept over its long history of development. However, there is no strong guarantee that certain optimizations will happen. Control over optimizations remains tricky, fragile and non-compositional. `INLINE` and `REWRITE` pragmas can be used to control code generation [GHC developers 2024a], but without strong guarantees, and their advanced usage requires knowledge of GHC internals. For example, correctly specifying the *ordering* of certain rule applications is often needed. We also have to care about formal function arities. Infamously, the function composition operator is defined as (.) f g = λ x → f (g x) in the base libraries, instead of as (.) f g x = f (g x), to get better inlining behavior [GHC developers 2024b]. It is common practice in high-performance Haskell programming to visually review GHC's optimized code output.

1.1 Closure-Free Staged Compilation

In this paper we use staged compilation to address issues of robustness. The idea is to shift as much as possible work from general-purpose optimization to metaprograms.

Metaprograms can be deterministic, transparent, and can be run efficiently, using fast interpreters or machine code compilation. In contrast, general-purpose optimizers are slower to run, less transparent and less robust. Also, metaprogramming allows library authors to exploit domain-specific optimizations, while it is not realistic for general-purpose optimizers to know about all domains.

On the other hand, metaprogramming requires some additional care and input from programmers. Historically, there have been problems with ergonomics as well. In weakly-typed staged systems, code generation might fail *too late* in the pipeline, producing incomprehensible errors. Or, tooling that works for an object language (like debugging, profiling, IDEs) may not work for metaprogramming, or metaprogramming may introduce heavy noise and boilerplate, obscuring the logic of programs and imposing restrictions on code structure.

Two-level type theory (2LTT) [Annenkov et al. 2019; Kovács 2022] makes it possible to use expressive dependent type theories for metaprogramming for a wide range of object languages. In this paper, we use 2LTT to sweeten the deal of staged compilation, aiming for a combination of strong guarantees, good ergonomics, high level of abstraction and easy-to-optimize code output.

Without optimizations we roughly get:

```
f = λ x. (>>=) monadReader (ask monadReaderReader) (λ b. if b
  then return monadReader (x + 10)
  else return monadReader (x + 20))
```

Here, monadReader and monadReaderReader are runtime dictionaries, respectively for the Monad and MonadReader instances, and, for example, (>>=) monadReader is a field projection from the dictionary. This results from the dictionary-passing elaboration of type classes [Wadler and Blott 1989]. We get a runtime closure from the λ b. ... function, and (>>=), ask and **return** also produce additional closures.

The difference between optimized and unoptimized code is already large here, and it gets even larger when we consider monad transformers or code that is polymorphic over monads. In Haskell, such code is pervasive, even in fairly basic programs which do not use fancy abstractions. Consider the mapM function from the Haskell Prelude:

```
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
```

This is a third-order rank-2 polymorphic function in disguise, because its monad dictionary argument contains the polymorphic second-order method (>>=). Compiling mapM efficiently relies on inlining the instance dictionary, then inlining the methods contained there, and also inlining the functions that the higher-order binding is applied to.

GHC's optimization efforts are respectable, and it has gotten quite adept over its long history of development. However, there is no strong guarantee that certain optimizations will happen. Control over optimizations remains tricky, fragile and non-compositional. `INLINE` and `REWRITE` pragmas can be used to control code generation [GHC developers 2024a], but without strong guarantees, and their advanced usage requires knowledge of GHC internals. For example, correctly specifying the *ordering* of certain rule applications is often needed. We also have to care about formal function arities. Infamously, the function composition operator is defined as (.) f g = λ x → f (g x) in the base libraries, instead of as (.) f g x = f (g x), to get better inlining behavior [GHC developers 2024b]. It is common practice in high-performance Haskell programming to visually review GHC's optimized code output.

1.1 Closure-Free Staged Compilation

In this paper we use staged compilation to address issues of robustness. The idea is to shift as much work as possible from general-purpose optimization to metaprograms.

Metaprograms can be deterministic, transparent, and can be run efficiently, using fast interpreters or machine code compilation. In contrast, general-purpose optimizers are slower to run, less transparent and less robust. Also, metaprogramming allows library authors to exploit domain-specific optimizations, while it is not realistic for general-purpose optimizers to know about all domains.

On the other hand, metaprogramming requires some additional care and input from programmers. Historically, there have been problems with ergonomics as well. In weakly-typed staged systems, code generation might fail *too late* in the pipeline, producing incomprehensible errors. Or, tooling that works for an object language (like debugging, profiling, IDEs) may not work for metaprogramming, or metaprogramming may introduce heavy noise and boilerplate, obscuring the logic of programs and imposing restrictions on code structure.

We use **two-level type theory** (2LTT) [Annenkov et al. 2019; Kovács 2022] to sweeten the deal of staged compilation, aiming for a combination of strong guarantees, good ergonomics, high level

We develop a particular two-level type theory for this purpose, which we call **CFTT**, short for "closure-free type theory". This consists of:

- A simply-typed object theory with first-order functions, general recursion and finitary algebraic data types. This language is easy to optimize and compile in the downstream pipeline, but it lacks many convenience features.
- A dependent type theory for the compile-time language. This allows us to recover many features by metaprogramming.

Since the object language is first-order, we guarantee that all programs in CFTT can be ultimately compiled without any dynamic closures, using only calls and jumps to statically known code. Why emphasize closures? They are the foundation to almost all abstraction tools in functional programming:

- Higher-order functions in essentially all functional languages are implemented with closures.
- Type classes in Haskell use dictionary-passing, which relies on closures for function methods [Wadler and Blott 1989].
- Functors and first-class modules in OCaml [Leroy et al. 2023] and other ML-s rely on closures.

Hence, doing functional programming without closures is a clear demonstration that we can get rid of abstraction overheads.

Perhaps surprisingly, little practical programming relies essentially on closures. Most of the time, programmers use higher-order functions for *abstraction*, such as when mapping over lists, where it is expected that the mapping function will be inlined. We note though that our setup is compatible with closures as well, and it can support two separate type formers for closure-based and non-closure-based "static" functions. Having both of these would be desirable in a practical system. In the current work we focus on the closure-free case because it is much less known and developed, and it is interesting to see how far we can go with it.

1.2 Contributions

- In Section 2 we present the two-level type theory CFTT, where the object level is first-order simply-typed and the meta level is dependently typed. The object language supports an operational semantics without runtime closures, and can be compiled with only statically known function calls. We provide a supplementary Agda formalization of the operational semantics of the object language.
- In Section 3 we build a monad transformer library. We believe that this is a good demonstration, because monads and monad transformers are the most widely used effect system in Haskell, and at the same time their compilation to efficient code can be surprisingly difficult. The main idea is to have monads and monad transformers *only* in the metalanguage, and try to express as much as possible at the meta level, and funnel the object-meta interactions through specific *binding-time improvements*. We show how almost all definitions from the Haskell ecosystem of monad transformers can be reused in the meta level of CFTT. We also develop let-insertion, join points and case switching on object-level data.
- In Section 4 we build a pull-based stream fusion library. Here, we demonstrate essential usage of dependent types, in providing guaranteed fusion for arbitrary combinations of concatMap and zip. We use a state machine representation that is based on *sums-of-products* of object-level values. We show that CFTT is compatible with a *generativity* axiom, which internalizes the fact that metaprograms cannot inspect the structure of object-level terms. We use this to show that the universe of sums-of-products is closed under Σ-types. This in turn enables a very concise definition of concatMap.

of abstraction and easy-to-optimize code output. We develop a particular two-level type theory for this purpose, which we call **CFTT**, short for "closure-free type theory". This consists of:

- A simply-typed object theory with first-order functions, general recursion and finitary algebraic data types. This language is easy to optimize and compile in the downstream pipeline, but it lacks many convenience features.
- A dependent type theory for the compile-time language. This allows us to recover many features by metaprogramming.

Since the object language is first-order, we guarantee that all programs in CFTT can be ultimately compiled without any dynamic closures, using only calls and jumps to statically known code. Why emphasize closures? They are the foundation to almost all abstraction tools in functional programming:

- Higher-order functions in essentially all functional languages are implemented with closures.
- Type classes in Haskell use dictionary-passing, which relies on closures for function methods [Wadler and Blott 1989].
- Functors and first-class modules in OCaml [Leroy et al. 2023] and other ML-s rely on closures.

Hence, doing functional programming without closures is a clear demonstration that we can get rid of abstraction overheads.

Perhaps surprisingly, little practical programming relies essentially on closures. Most of the time, programmers use higher-order functions for *abstraction*, such as when mapping over lists, where it is expected that the mapping function will be inlined. We note though that our setup is compatible with closures as well, and it can support two separate type formers for closure-based and non-closure-based "static" functions. Having both of these would be desirable in a practical system. In the current work we focus on the closure-free case because it is much less known and developed, and it is interesting to see how far we can go with it.

1.2 Contributions

- In Section 2 we present the two-level type theory CFTT, where the object level is first-order simply-typed and the meta level is dependently typed. The object language supports an operational semantics without runtime closures, and can be compiled with only statically known function calls. We provide a supplementary Agda formalization of the operational semantics of the object language.
- In Section 3 we build a staged library for monad transformers [Liang et al. 1995]. We believe that this is a good demonstration, because monads and monad transformers are the most widely used effect system in Haskell, and at the same time their compilation to efficient code can be surprisingly difficult. The continuation monad is well-known in staged compilation [Bondorf 1992], and staged state monads have also been used [Carette and Kiselyov 2011; Kiselyov et al. 2004; Swadi et al. 2006]. These works used specific monads as tools for domain-specific code generation. In contrast, we propose ubiquitous staging for general-purpose monadic programming, where users can write code that looks similar to monadic code in Haskell but with deterministic and robust compilation to efficient code.
- In Section 4 we build a pull-based stream fusion library. Here, we demonstrate essential usage of dependent types, in providing guaranteed fusion for arbitrary combinations of concatMap and zip. We use a state machine representation that is based on *sums-of-products* of object-level values. We show that CFTT is compatible with a *generativity* axiom, which internalizes the fact that metaprograms cannot inspect the structure of object-level terms. We use this to show that the universe of sums-of-products is closed under Σ-types. This in turn enables a very concise definition of concatMap.

- We adapt the contents of the paper to typed Template Haskell [Xie et al. 2022], with some modifications, simplifications and fewer guarantees about generated code. In particular, Haskell does not have enough dependent types for the simple concatMap definition, but we can still work around this limitation. We also provide a precise Agda embedding of CFTT and our libraries as described in this paper. Here, the object theory is embedded as a collection of postulated operations, and we can use Agda's normalization command to print out generated object code.

2 OVERVIEW OF CFTT

In the following we give an overview of CFTT features. We first review the meta-level language, then the object-level one, and finally the staging operations which bridge between the two.

2.1 The Meta Level

MetaTy is the universe of types in the compile-time language. We will often use the term "metatype" to refer to inhabitants of MetaTy, and use "metaprogram" for inhabitants of metatypes. MetaTy supports dependent functions, Σ-types and indexed inductive types [Dybjer 1994].

Formally, MetaTy is additionally indexed by universes levels (orthogonally to staging), and we have $\mathsf{MetaTy}_i : \mathsf{MetaTy}_{i+1}$. However, universe levels add noise and they are not too relevant to the current paper, so we will omit them.

Throughout this paper we use a mix of Agda and Haskell syntax for CFTT. Dependent functions and implicit arguments follow Agda. A basic example:

$$\mathsf{id} : \{A : \mathsf{MetaTy}\} \to A \to A$$
$$\mathsf{id} = \lambda\, x.\, x$$

Here, the type argument is implicit, and it gets inferred when we use the function. For example, id True is elaborated to id {Bool} True, where the braces mark an explicit application for the implicit argument. Inductive types can be introduced using a Haskell-like ADT notation, or with a GADT-style one:

$$\mathbf{data}\ \mathsf{Bool}_\mathsf{M} : \mathsf{MetaTy} = \mathsf{True}_\mathsf{M} \mid \mathsf{False}_\mathsf{M}$$

$$\mathbf{data}\ \mathsf{Bool}_\mathsf{M} : \mathsf{MetaTy}\ \mathbf{where}\quad \mathsf{True}_\mathsf{M} : \mathsf{Bool}_\mathsf{M}\quad \mathsf{False}_\mathsf{M} : \mathsf{Bool}_\mathsf{M}$$

Note that we added an $_\mathsf{M}$ subscript to the type; when analogous types can be defined both on the meta and object levels, we will sometimes use this subscript to disambiguate the meta-level version.

We use Haskell-like newtype notation, such as in $\mathbf{newtype}\ \mathsf{Wrap}\ A = \mathsf{Wrap}\ \{\mathsf{unWrap} : A\}$, and also a similar notation for (dependent) record types, for instance as in

$$\mathbf{data}\ \mathsf{Record} = \mathsf{Record}\ \{\mathsf{field1} : A,\ \mathsf{field2} : B\}.$$

All construction and elimination rules for type formers in MetaTy stay within MetaTy. For example, induction on meta-level values can only produce meta-level values.

2.2 The Object Level

Ty is the universe of types in the object language. It is itself a metatype, so so we have Ty : MetaTy. All construction and elimination rules of type formers in Ty stay within Ty. We further split Ty to two sub-universes.

First, **ValTy** : MetaTy is the universe of *value types*. ValTy supports parameterized algebraic data types, where parameters can have arbitrary types, but all constructor field types must be in ValTy.

- We adapt the contents of the paper to typed Template Haskell [Xie et al. 2022], with some modifications, simplifications and fewer guarantees about generated code. In particular, Haskell does not have enough dependent types for the simple concatMap definition, but we can still work around this limitation. We also provide a precise Agda embedding of CFTT and our libraries as described in this paper. Here, the object theory is embedded as a collection of postulated operations, and we can use Agda's normalization command to print out generated object code.

2 OVERVIEW OF CFTT

In the following we give an overview of CFTT features. We first review the meta-level language, then the object-level one, and finally the staging operations which bridge between the two.

2.1 The Meta Level

MetaTy is the universe of types in the compile-time language. We will often use the term "metatype" to refer to inhabitants of MetaTy, and use "metaprogram" for inhabitants of metatypes. MetaTy supports dependent functions, Σ-types and indexed inductive types [Dybjer 1994].

Formally, MetaTy is additionally indexed by universe levels (orthogonally to staging), and we have $\mathsf{MetaTy}_i : \mathsf{MetaTy}_{i+1}$. However, universe levels add noise and they are not too relevant to the current paper, so we will omit them.

Throughout this paper we use a mix of Agda and Haskell syntax for CFTT. Dependent functions and implicit arguments follow Agda. A basic example:

$$\mathsf{id} : \{A : \mathsf{MetaTy}\} \to A \to A$$
$$\mathsf{id} = \lambda\, x.\, x$$

Here, the type argument is implicit, and it gets inferred when we use the function. For example, id True is elaborated to id {Bool} True, where the braces mark an explicit application for the implicit argument. Inductive types can be introduced using a Haskell-like ADT notation, or with a GADT-style one:

$$\mathbf{data}\ \mathsf{Bool}_\mathsf{M} : \mathsf{MetaTy} = \mathsf{True}_\mathsf{M} \mid \mathsf{False}_\mathsf{M}$$

$$\mathbf{data}\ \mathsf{Bool}_\mathsf{M} : \mathsf{MetaTy}\ \mathbf{where}\quad \mathsf{True}_\mathsf{M} : \mathsf{Bool}_\mathsf{M}\quad \mathsf{False}_\mathsf{M} : \mathsf{Bool}_\mathsf{M}$$

Note that we added an $_\mathsf{M}$ subscript to the type; when analogous types can be defined both on the meta and object levels, we will sometimes use this subscript to disambiguate the meta-level version.

We use Haskell-like newtype notation, such as in $\mathbf{newtype}\ \mathsf{Wrap}\ A = \mathsf{Wrap}\ \{\mathsf{unWrap} : A\}$, and also a similar notation for (dependent) record types, for instance as in

$$\mathbf{data}\ \mathsf{Record} = \mathsf{Record}\ \{\mathsf{field1} : A,\ \mathsf{field2} : B\}.$$

All construction and elimination rules for type formers in MetaTy stay within MetaTy. For example, induction on meta-level values can only produce meta-level values.

2.2 The Object Level

Ty is the universe of types in the object language. It is itself a metatype, so we have Ty : MetaTy. All construction and elimination rules of type formers in Ty stay within Ty. We further split Ty to two sub-universes.

First, **ValTy** : MetaTy is the universe of *value types*. ValTy supports parameterized algebraic data types, where parameters can have arbitrary types, but all constructor field types must be in ValTy.

Since ValTy is a sub-universe of Ty, we have that when A : ValTy then also A : Ty. Formally, this is specified as an explicit embedding operation, but we will use implicit subtyping for convenience.

Second, **CompTy** : MetaTy is the universe of *computation types*. This is also a sub-universe of Ty with implicit coercions. For now, we only specify that CompTy contains functions whose domains are value types:

$$- \to - : \mathsf{ValTy} \to \mathsf{Ty} \to \mathsf{CompTy}$$

For instance, if Bool : ValTy is defined as an object-level ADT, then Bool → Bool : CompTy, hence also Bool → Bool : Ty. However, (Bool → Bool) → Bool is ill-formed, since the domain is not a value type. Let us look at an example for an object-level program, where we already have natural numbers declared as **data** Nat := Zero | Suc Nat:

```
add : Nat → Nat → Nat
add := letrec go n m := case n of
         Zero  → m;
         Suc n → Suc (go n m);
       go
```

Recursive definitions are introduced with **letrec** . The general syntax is **letrec** x : A := t; u, where the A type annotation can be omitted. **letrec** can be only used to define computations, not values (hence, only functions can be recursive so far).

Object-level definitions use := as notation, instead of the = that is used for meta-level ones. We also have non-recursive **let**, which can be used to define computations and values alike, and can be used to shadow binders:

```
f : Nat → Nat
f x := let x := x + 10; let x := x + 20; x * 10
```

We also allow **newtype** definitions, both in ValTy and CompTy. These are assumed to be erased at runtime. In the Haskell implementation they are important for guiding type class resolution, and we think that the explicit wrapping makes many definitions more comprehensible in CFTT as well.

Values are call-by-value at runtime; they are computed eagerly in function applications and **let**-s. **let**-definitions can be used to define inhabitants of any type, and the type of the **let** body can be also arbitrary. Additionally, the right hand sides of **case** branches can also have arbitrary types. So the following is well-formed:

```
f : Bool → Nat → Nat
f b := case b of True → (λ x. x + 10); False → (λ x. x * 10)
```

In contrast, computations are call-by-name, and the only way we can compute with functions is to apply them to value arguments. The call-by-name strategy is fairly benign here and does not lead to significant duplication of computation, because functions cannot escape their scope; they cannot be passed as arguments or stored in data constructors. This makes it possible to run object programs without using dynamic closures. This point is not completely straightforward; consider the previous f function which has λ-expressions under a **case**.

However, the call-by-name semantics lets us transform f to λ b x. **case** b **of** True → x+10; False → x * 10, and more generally we can transform programs so that every function call becomes *saturated*. This means that every function call is of the form f t_1 t_2 ... t_n, where f is a function variable and the definition of f immediately λ-binds *n* arguments. We do not detail this here. We provide formal syntax and operational semantics of the object language in the Agda supplement. We formalized the

Closure-Free Functional Programming in a Two-Level Type Theory 5

Since ValTy is a sub-universe of Ty, we have that when A : ValTy then also A : Ty. Formally, this is specified as an explicit embedding operation, but we will use implicit subtyping for convenience.

Second, **CompTy** : MetaTy is the universe of *computation types*. This is also a sub-universe of Ty with implicit coercions. For now, we only specify that CompTy contains functions whose domains are value types:

$$- \to - : \mathsf{ValTy} \to \mathsf{Ty} \to \mathsf{CompTy}$$

For instance, if Bool : ValTy is defined as an object-level ADT, then Bool → Bool : CompTy, hence also Bool → Bool : Ty. However, (Bool → Bool) → Bool is ill-formed, since the domain is not a value type. Let us look at an example for an object-level program, where we already have natural numbers declared as **data** Nat := Zero | Suc Nat:

```
add : Nat → Nat → Nat
add := letrec go n m := case n of
         Zero  → m;
         Suc n → Suc (go n m);
       go
```

Recursive definitions are introduced with **letrec** . The general syntax is **letrec** x : A := t; u, where the A type annotation can be omitted. **letrec** can be only used to define computations, not values (hence, only functions can be recursive so far).

Object-level definitions use := as notation, instead of the = that is used for meta-level ones. We also have non-recursive **let**, which can be used to define computations and values alike, and can be used to shadow binders:

```
f : Nat → Nat
f x := let x := x + 10; let x := x + 20; x * 10
```

We also allow **newtype** definitions, both in ValTy and CompTy. These are assumed to be erased at runtime. In the Haskell and Agda implementations they are important for guiding instance resolution, and we think that the explicit wrapping makes many definitions more comprehensible in CFTT as well.

Values are call-by-value at runtime; they are computed eagerly in function applications and **let**-s. **let**-definitions can be used to define inhabitants of any type, and the type of the **let** body can be also arbitrary. Additionally, the right hand sides of **case** branches can also have arbitrary types. So the following is well-formed:

```
f : Bool → Nat → Nat
f b := case b of True → (λ x. x + 10); False → (λ x. x * 10)
```

In contrast, computations are call-by-name, and the only way we can compute with functions is to apply them to value arguments. The call-by-name strategy is fairly benign here and does not lead to significant duplication of computation, because functions cannot escape their scope; they cannot be passed as arguments or stored in data constructors. This makes it possible to run object programs without using dynamic closures. This point is not completely straightforward; consider the previous f function which has λ-expressions under a **case**.

However, the call-by-name semantics lets us transform f to λ b x. **case** b **of** True → x+10; False → x * 10, and more generally we can transform programs so that every function call becomes *saturated*. This means that every function call is of the form f t_1 t_2 ... t_n, where f is a function variable and the definition of f immediately λ-binds *n* arguments. We do not detail this here. We provide formal syntax and operational semantics of the object language in the Agda supplement. We formalized the

specific translation steps that are involved in call saturation, but only specified the full translation informally.

2.2.1 Object-level definitional equality. This is a distinct notion from runtime semantics. Object programs are embedded in CFTT, which is a dependently typed language, so sometimes we need to decide definitional equality of object programs during type checking. The setup is simple: we have no β or η rules for object programs at all, nor any rule for **let**-unfolding. The main reason is the following: we care about the size and efficiency of generated code, and these properties are not stable under $\beta\eta$-conversion and **let**-unfolding. Moreover, since the object language has general recursion, we do not have a sensible and decidable notion of program equivalence anyway.

2.2.2 Comparison to call-by-push-value. We took inspiration from call-by-push-value (CBPV) [Levy 1999], and there are similarities to our object language, but there are also significant differences. Both systems have a value-computation distinction, with call-by-name computations and call-by-value values. However, our object theory supports variable binding at arbitrary types while CBPV only supports value variables. In CBPV, a let-definition for a function is only possible by first packing it up as a closure value (or "thunk"), which clearly does not suit our applications. Investigating the relation between CBPV and our object language could be future work.

2.3 Staging

With what we have seen so far, there is no interaction between the meta and object levels. We make such interaction possible with *staging operations*.

- For A : Ty, we have ⇑A : MetaTy, pronounced as "lift A". This is the type of metaprograms that produce A-typed object programs.
- For A : Ty and t : A, we have ⟨t⟩ : ⇑A, pronounced "quote t". This is the metaprogram which immediately returns t.
- For t : ⇑A, we have ∼t : *A*, pronounced "splice t". This inserts the result of a metaprogram into an object term. *Notation:* splicing binds stronger than function application, so f ∼x is parsed as f (∼x). We borrow this notation from MetaML [Taha and Sheard 2000].
- We have ⟨∼t⟩ ≡ t and ∼⟨t⟩ ≡ t as definitional equalities.

A CFTT program is a mixture of object-level and meta-level top-level definitions and declarations. *Staging* means running all metaprograms in splices and inserting their output into object code, keeping all object-level top entries and discarding all meta-level ones. Thus, staging takes a CFTT program as input, and produces output which is purely in the object-level fragment, with no metatypes and metaprograms remaining. The output is guaranteed to be well-typed.

Let us look at some basic staging examples. Recall the meta-level identity function; it can be used at the object-level too, by applying it to quoted terms:

$$\textbf{let}\ \mathsf{n} : \mathsf{Nat} := \sim(\mathsf{id}\ \langle 10 + 10\rangle); \ldots$$

Here, id is used at type ⇑Nat. During staging, the expression in the splice is evaluated, so we get ∼⟨10 + 10⟩, which is definitionally the same as 10 + 10, which is our staging output here. Boolean short-circuiting is another basic use-case:

$$\mathsf{and} : \Uparrow\mathsf{Bool} \to \Uparrow\mathsf{Bool} \to \Uparrow\mathsf{Bool}$$
$$\mathsf{and}\ \mathsf{x}\ \mathsf{y} = \langle \textbf{case}\ {\sim}\mathsf{x}\ \textbf{of}\ \mathsf{True} \to {\sim}\mathsf{y}; \mathsf{False} \to \mathsf{False}\rangle$$

Since the *y* expression is inlined under a **case** branch at every use site, it is only computed at runtime when *x* evaluates to True. In many situations, staging can be used instead of laziness to implement short-circuiting, and often with better runtime performance, avoiding the overhead of

specific translation steps that are involved in call saturation, but only specified the full translation informally.

Why not just make the object language less liberal, e.g. by disallowing λ under **case** or **let**, thereby making call saturation easier or more obvious? There is a trade-off between making the object language more restricted, and thus easier to compile, and making metaprogramming more convenient. We will see that the ability to insert **let**-s without restriction is very convenient in code generation, and likewise the ability to have arbitrary object expressions in **case** bodies. In this paper we go with the most liberal object syntax, at the cost of needing more downstream processing.

2.2.1 Object-level definitional equality. This is a distinct notion from runtime semantics. Object programs are embedded in CFTT, which is a dependently typed language, so sometimes we need to decide definitional equality of object programs during type checking. The setup is simple: we have no β or η rules for object programs at all, nor any rule for **let**-unfolding. The main reason is the following: we care about the size and efficiency of generated code, and these properties are not stable under $\beta\eta$-conversion and **let**-unfolding. Moreover, since the object language has general recursion, we do not have a sensible and decidable notion of program equivalence anyway.

2.2.2 Comparison to call-by-push-value. We took inspiration from call-by-push-value (CBPV) [Levy 1999], and there are similarities to our object language, but there are also significant differences. Both systems have a value-computation distinction, with call-by-name computations and call-by-value values. However, our object theory supports variable binding at arbitrary types while CBPV only supports value variables. In CBPV, a let-definition for a function is only possible by first packing it up as a closure value (or "thunk"), which clearly does not suit our applications. Investigating the relation between CBPV and our object language could be future work.

2.3 Staging

With what we have seen so far, there is no interaction between the meta and object levels. We make such interaction possible with the following primitives.

- For A : Ty, we have ⇑A : MetaTy, pronounced as "lift A". This is the type of metaprograms that produce A-typed object programs.
- For A : Ty and t : A, we have ⟨t⟩ : ⇑A, pronounced "quote t". This is the metaprogram which immediately returns t.
- For t : ⇑A, we have ∼t : *A*, pronounced "splice t". This inserts the result of a metaprogram into an object term. *Notation:* splicing binds stronger than function application, so f ∼x is parsed as f (∼x). We borrow this from MetaML [Taha and Sheard 2000].
- We have ⟨∼t⟩ ≡ t and ∼⟨t⟩ ≡ t as definitional equalities.

We use **unstaging** to refer to the process of extracting object code from CFTT programs, by evaluating all metaprograms in splices. This term has been sporadically used in the literature, e.g. by Odersky and Rompf [Rompf and Odersky 2012] and also by Choi et al. [Choi et al. 2011] with a somewhat different meaning in a multi-stage context. The main precursor to this paper used "staging" instead of our "unstaging" [Kovács 2022]; we use the latter because the former conflicts with other common usages of "staging".

Let us look at some basic examples. Recall the meta-level identity function; it can be used at the object-level too, by applying it to quoted terms:

$$\textbf{let}\ \mathsf{n} : \mathsf{Nat} := \sim(\mathsf{id}\ \langle 10 + 10\rangle); \ldots$$

Here, id is used at type ⇑Nat. During unstaging, the expression in the splice is evaluated, so we get ∼⟨10 + 10⟩, which is definitionally the same as 10 + 10, which is our code output here. Boolean

thunking. Consider the map function now:

```
map : {A B : ValTy} → (⇑A → ⇑B) → ⇑(List A) → ⇑(List B)
map f as = ⟨letrec go as := case as of
                Nil       → Nil;
                Cons a as → Cons ~(f ⟨a⟩) (go as);
           go ~as⟩
```

For example, this can be used as **let** f as : List Nat → List Nat := ~(map (λ x. ⟨~x + 10⟩) ⟨as⟩). This is staged to a recursive definition where the mapping function is inlined into the Cons case as Cons a as → Cons (a + 10) (go as). Note that map has to abstract over value types, since lists can only contain values, not functions. Also, the mapping function has type ⇑A → ⇑B, instead of ⇑(A → B). The former type is often preferable to the latter in staging; the former is a metafunction with useful computational content, while the latter is merely a black box that computes object code. If we have f : ⇑(A → B), and f is staged to ⟨λ x. t⟩, then ~f u is staged to an undesirable "administrative" β-redex (λ x. t) u.

2.4 Semantics of Staging

We give a short summary of the semantics of staging here. We reuse most of the semantics from [Kovács 2022] and only mention additions and differences. The reader may refer to ibid. for technical details.

Staging is given by evaluation of CFTT into a presheaf model, where the base category is given as follows: objects are typing contexts of the object language and morphisms are parallel substitutions. The equality of morphisms is given by strict syntactic equality of object terms. Ty is interpreted as the presheaf which is constantly the set of object-theoretic types. Soundness of staging is proven by a logical relation model, internally to the mentioned presheaf model. Soundness means that the output of staging, viewed as a CFTT term, is definitionally equal to the input.

The main addition in CFTT is the assumption of all inductive families in the meta-level. However, based on the results in [Hugunin 2020], it is enough to extend the meta-level syntax in [Kovács 2022] with W-types and identity types to derive all inductive families. We consider the semantics of these in the following.

2.4.1 W-types. W-types have a standard object-wise inductive definition in the presheaf model [Moerdijk and Palmgren 2000]. Additionally, we need to give a logical relation interpretation to W-types, as required for the soundness proof for staging. In Section 5.2. of [Kovács 2022] a serialization map is used to define the relation for natural numbers, however, there is no analogous map for W-types because we cannot build finite terms from infinitely branching trees.

Instead, we give an inductive definition for the logical relation at W-types. First, assuming $A : \widehat{\mathsf{Set}}$ and $B : A \to \widehat{\mathsf{Set}}$, we have the usual constructor for the semantic W-type in the internal language of the presheaf model, as $\mathsf{sup} : (a : A) \to (B\,a \to \mathsf{W}\,A\,B) \to \mathsf{W}\,A\,B$. Second, assuming $A : \mathsf{Ty}\,\Gamma$ and $B : \mathsf{Ty}\,(\Gamma \triangleright A)$ in CFTT, we internalize the sup constructor which constructs a term of a syntactic W-type from restricted terms:

$$\mathbb{R}_{\mathsf{sup}} : (\alpha : \mathbb{R}\,A\,\gamma) \to \mathbb{R}\,(B \Rightarrow (\mathsf{W}_{\mathsf{CFTT}}\,A\,B)[\mathsf{wk}])\,(\gamma,\, \alpha) \to \mathbb{R}\,(\mathsf{W}_{\mathsf{CFTT}}\,A\,B)\,\gamma$$

Here, $\mathsf{W}_{\mathsf{CFTT}}$ denotes the W-type former in CFTT, $B \Rightarrow (\mathsf{W}_{\mathsf{CFTT}}\,A\,B)[\mathsf{wk}]$ is a non-dependent function type in CFTT, and wk is a weakening substitution.

Then, assuming $\gamma^{\circ} : \Gamma^{\circ}\,\gamma\,\gamma'$, we inductively define a relation W° between $\mathsf{W}\,(\mathbb{E}\,A\,\gamma)\,(\lambda\,\alpha.\,\mathbb{E}\,B\,(\gamma,\,\alpha))$ and $\mathbb{R}\,(\mathsf{W}_{\mathsf{CFTT}}\,A\,B)\,\gamma'$, which relates $\mathsf{sup}\,\alpha\,f$ to $\mathbb{R}_{\mathsf{sup}}\,\alpha'\,f'$ if α is related to α' at type A and f is

short-circuiting is another basic use-case:

```
and : ⇑Bool → ⇑Bool → ⇑Bool
and x y = ⟨case ~x of True → ~y; False → False⟩
```

Since the y expression is inlined under a **case** branch at every use site, it is only computed at runtime when x evaluates to True. In many situations, staging can be used instead of laziness to implement short-circuiting, and often with better runtime performance, avoiding the overhead of thunking. Consider the map function now:

```
map : {A B : ValTy} → (⇑A → ⇑B) → ⇑(List A) → ⇑(List B)
map f as = ⟨letrec go as := case as of
                Nil       → Nil;
                Cons a as → Cons ~(f ⟨a⟩) (go as);
           go ~as⟩
```

For example, this can be used as **let** f as : List Nat → List Nat := ~(map (λ x. ⟨~x + 10⟩) ⟨as⟩). This is unstaged to a recursive definition where the mapping function is inlined into the Cons case as Cons a as → Cons (a + 10) (go as). Note that map has to abstract over value types, since lists can only contain values, not functions. Also, the mapping function has type ⇑A → ⇑B, instead of ⇑(A → B). The former type is often preferable to the latter; the former is a metafunction with useful computational content, while the latter is merely a black box that computes object code. If we have f : ⇑(A → B), and f computes to ⟨λ x. t⟩, then ~f u is unstaged to an undesirable "administrative" β-redex (λ x. t) u.

2.3.1 Comparison to defunctionalization. Defunctionalization is a program translation which represents higher-order functions using only first-order functions and inductive data [Danvy and Nielsen 2001; Reynolds 1998]. The idea is to represent each closure as a constructor of an inductive type, and implement closure application as a top-level first-order function which switches on all possible closure constructors. Hence, like our unstaging algorithm, defunctionalization produces first-order code from higher-order code. We summarize the differences.

- Defunctionalization mainly supports *compiler optimizations*, by making closures transparent to analysis. It is not an optimization by itself; it does not change the amount of dynamic control flow or dynamic allocations in a program. A dynamic closure call becomes a dynamic case-switch and a heap-allocated "functional" closure becomes a heap-allocated inductive constructor.
- Staging mainly supports optimization by *programmers* by providing control over code generation. Unstaging does not translate higher-order functions to anything in particular. Instead, unstaging is the execution of higher-order metaprograms which produce first-order programs.

2.4 Staged Semantics of CFTT

To obtain an unstaging algorithm, together with its correctness, the main task is to extend [Kovács 2022] with identity types and inductive families in the metalanguage, since we use those types in this paper. For inductive types, it is the most sensible to add W-types, since all inductive families can be faithfully derived from them in our setting [Hugunin 2020].

This however requires a substantial amount of technical background from [Kovács 2022], so we relegate it to Appendix A. Here we only give an overview.

related to f' at type $B[\alpha'] \Rightarrow \mathrm{W}_{\mathrm{CFTT}}\, A\, B$. The relation at the non-dependent function type $- \Longrightarrow -$ is given as pointwise preservation of relations. With this, we can also give relational interpretation to the syntactic sup rule in CFTT, and we can use induction on W" to interpret the syntactic induction rule for W. Thus, we get soundness of staging with W-types.

2.4.2 Identity type. Our treatment of the identity type is influenced by the following: we would like to assume equations as axioms in CFTT, which are true in the presheaf model (i.e. they are true at staging-time), without blocking staging as a computation. In Section 4.3 we use such an axiom to good effect. However, axioms do block computation in CFTT up to definitional equality, which implies that the soundness of staging fails. Our solution:

- Have a standard intensional identity type in CFTT.
- Before staging, translate the intensional identity to extensional identity. This amounts to erasing all identity proofs and transports.
- Perform staging after erasure of identity proofs, i.e. model extensional identity in the semantics.

In the presheaf model, we interpret the extensional identity in a standard way, as strict equality of sections of presheaves, and soundness for extensional identity becomes trivial to show. Hence, the soundness property for our overall staging pipeline becomes the following: the staging output is definitionally equal to the identity-erased version of the staging input.

3 MONADS & MONAD TRANSFORMERS

In this section we build a library for monads and monad transformers. We believe that this is a good demonstration of CFTT-s abilities, since monads are ubiquitous in Haskell programming and they also introduce a great amount of abstraction that should be optimized away.

3.1 Binding-Time Improvements

We start with some preparatory work before getting to monads. We saw that $\Uparrow A \to \Uparrow B$ is usually preferable to $\Uparrow(A \to B)$. The two types are actually equivalent during staging, up to the runtime equivalence of object programs, and we can convert back and forth in CFTT:

$$\mathbf{up} : \Uparrow(A \to B) \to \Uparrow A \to \Uparrow B \qquad \mathbf{down} : (\Uparrow A \to \Uparrow B) \to \Uparrow(A \to B)$$
$$\mathbf{up}\, f\, a = \langle \sim f\, \sim a \rangle \qquad \mathbf{down}\, f = \langle \lambda\, a.\, \sim(f\, \langle a \rangle) \rangle$$

We cannot show internally, using propositional equality, that these functions are inverses, since we do not have $\beta\eta$-rules for object functions; but we will not need this proof in the rest of the paper.

In the staged compilation and partial evaluation literature, the term *binding time improvement* is used to refer to such conversions, where the "improved" version supports more compile-time computation [Jones et al. 1993]. A general strategy for generating efficient "fused" programs, is to try to work as much as possible with improved representations, and only convert back to object code at points where runtime dependencies are unavoidable. Let us look at binding-time-improvement for product types now:

$$\mathbf{up} : \Uparrow(A, B) \to (\Uparrow A, \Uparrow B) \qquad \mathbf{down} : (\Uparrow A, \Uparrow B) \to \Uparrow(A, B)$$
$$\mathbf{up}\, x = (\langle \mathrm{fst}\, \sim x \rangle, \langle \mathrm{snd}\, \sim x \rangle) \qquad \mathbf{down}\, (x, y) = \langle (\sim x, \sim y) \rangle$$

Here we overload Haskell-style product type notation, both for types and the pair constructor, at both levels. There is a problem with this conversion though: **up** uses $x : \Uparrow(A, B)$ twice, which can increase code size and duplicate runtime computations. For example, **down** (**up** $\langle f\, x \rangle$) is staged to $\langle (\mathrm{fst}\, (f\, x), \mathrm{snd}\, (f\, x)) \rangle$. It would be safer to first let-bind an expression with type $\Uparrow(A, B)$, and then only use projections of the newly bound variable. This is called *let-insertion* in staged compilation.

- The input of unstaging is a CFTT term with an object-level type, which only depends on object-level free variables. The output is a term in the object theory. In the other direction, there is an evident embedding of object-theoretic terms as CFTT terms.
- *Soundness* means that unstaging followed by embedding is the identity map, up to conversion. In other words, unstaging respects conversion of CFTT terms.
- *Stability* means that embedding followed by unstaging is the identity map. In other words, unstaging has no action on terms which are already purely object-level (i.e. contain no splices).

In [Kovács 2022], there is one more property, called *strictness*, which means that unstaging does not perform $\beta\eta$-conversion in the object theory, but this holds trivially in our case since we do not have such conversion rules.

However, for CFTT we slightly depart from the above notion of soundness, for the following reason. We would like to assume certain propositional identities as *axioms* in the metalanguage, without blocking unstaging as a computation. We use such an axiom to good effect in Section 4.3.

Axioms clearly block computation, so they are incompatible with the mentioned notion of soundness. But we are only interested in equations which hold definitionally during unstaging. Hence, we can do the following: first, we erase all identity proofs and their transports from CFTT programs, then we proceed with unstaging as in [Kovács 2022]. This erasure can be formalzied as a syntactic translation [Boulier et al. 2017] from CFTT to CFTT extended with the equality reflection principle, which says that propositional equality implies definitional equality (see e.g. [Hofmann 1995]). Thus, soundness of staging for CFTT holds up to erasure of identity proofs.

3 MONADS & MONAD TRANSFORMERS

In this section we build a library for monads and monad transformers. We believe that this is a good demonstration of CFTT's abilities, since monads are ubiquitous in Haskell programming and they also introduce a great amount of abstraction that should be optimized away.

3.1 Binding-Time Improvements

We do a preliminary overview before getting to monads. In the staged compilation and partial evaluation literature, the term *binding time improvement* is used to refer to such conversions, where the "improved" version supports more compile-time computation [Jones et al. 1993, Chapter 12].

Translation from $\Uparrow A \to \Uparrow B$ to $\Uparrow(A \to B)$ is a basic binding-time improvement, as an η-expansion [Danvy et al. 1996]. The two types are equivalent during unstaging, up to the runtime equivalence of object programs, and we can convert back and forth in CFTT, the same way as in MetaML [Taha and Sheard 2000, Section 9]:

$$\mathbf{up} : \Uparrow(A \to B) \to \Uparrow A \to \Uparrow B \qquad \mathbf{down} : (\Uparrow A \to \Uparrow B) \to \Uparrow(A \to B)$$
$$\mathbf{up}\, f\, a = \langle \sim f\, \sim a \rangle \qquad \mathbf{down}\, f = \langle \lambda\, a.\, \sim(f\, \langle a \rangle) \rangle$$

We cannot show internally, using propositional equality, that these functions are inverses, since we do not have $\beta\eta$-rules for object functions; but we will not need this proof in the rest of the paper.

A general strategy for generating efficient "fused" programs, is to try to work as much as possible with improved representations, and only convert back to object code at points where runtime dependencies are unavoidable. Let us look at binding-time-improvement for product types now:

$$\mathbf{up} : \Uparrow(A, B) \to (\Uparrow A, \Uparrow B) \qquad \mathbf{down} : (\Uparrow A, \Uparrow B) \to \Uparrow(A, B)$$
$$\mathbf{up}\, x = (\langle \mathrm{fst}\, \sim x \rangle, \langle \mathrm{snd}\, \sim x \rangle) \qquad \mathbf{down}\, (x, y) = \langle (\sim x, \sim y) \rangle$$

But it is impossible to use let-insertion in **up** because the return type is in MetaTy, and we cannot introduce object binders in meta-level code. Fortunately, there is a principled solution.

3.2 The Code Generation Monad

The solution is to use a monad which extends MetaTy with the ability to freely generate object-level code and introduce object binders.

```
newtype Gen (A : MetaTy) = Gen {unGen : {R : Ty} → (A → ⇑R) → ⇑R}
```

This is a monad in MetaTy in a standard sense:

```
instance Monad Gen where
  return a = Gen $ λ k. k a
  ga >>= f  = Gen $ λ k. unGen ga (λ a. unGen (f a) k))
```

From now on, we reuse Haskell-style type classes and **do**-notation in CFTT. We will use type classes in an informal way, without precisely specifying how they work. However, type classes are used in essentially the same way in the Agda and Haskell implementations, with modest technical differences.

From the Monad instance, the Functor and Applicative instances can be also derived. We reuse (<$>) and (<*>) for applicative notation as well. An inhabitant of Gen A can be viewed as an action whose effect is to produce some object code. Gen can be only "run" when the result type is an object type:

```
runGen : Gen (⇑A) → ⇑A
runGen ma = unGen ma id
```

We can let-bind object expressions in Gen:

```
gen : {A : Ty} → ⇑A → Gen (⇑A)
gen a = Gen $ λ k. ⟨let x : A := ~a; ~(k ⟨x⟩)⟩
```

And also recursive definitions of computations:

```
genRec : {A : CompTy} → (⇑A → ⇑A) → Gen (⇑A)
genRec f = Gen $ λ k. ⟨letrec x : A := ~(f ⟨x⟩); ~(k ⟨x⟩)⟩
```

Now, using do-notation, we may write **do** {x ← gen ⟨10 + 10⟩; y ← gen ⟨20 + 20⟩}; **return** ⟨x + y⟩}, for a Gen (⇑Nat) action. Running this with runGen yields ⟨**let** x := 10 + 10; **let** y := 20 + 20; x + y⟩. We can also define a "safer" binding-time improvement for products, using let-insertion:

```
up : ⇑(A, B) → Gen (⇑A, ⇑B)
up x = do x ← gen x
          return (⟨fst ~x⟩, ⟨snd ~x⟩)

down : Gen (⇑A, ⇑B) → ⇑(A, B)
down x = do (a, b) ← x
            return ⟨(~a, ~b)⟩
```

Working in Gen is convenient, since we can freely generate object code and also have access to the full metalanguage. Also, the point of staging is that *eventually* all metaprograms will be used for the purpose of code generation, so we eventually runGen all of our actions. So why not just always work in Gen? The implicit nature of Gen may make it harder to reason about the size and content of generated code. This is a bit similar to the IO monad in Haskell, where eventually everything needs to run in IO, but we may not want to write all of our code in IO.

Here we overload Haskell-style product type notation, both for types and the pair constructor, at both levels. There is a problem with this conversion though: **up** uses x : ⇑(A, B) twice, which can increase code size and duplicate runtime computations. For example, **down** (**up** ⟨f x⟩) is staged to ⟨(fst (f x), snd (f x))⟩. It would be safer to first let-bind an expression with type ⇑(A, B), and then only use projections of the newly bound variable. This is called *let-insertion* in staged compilation. But it is impossible to use let-insertion in **up** because the return type is in MetaTy, and we cannot introduce object binders in meta-level code.

3.2 The Code Generation Monad

A principled solution to the previous issue is to write code generators in continuation-passing style, as first proposed by Bondorf [Bondorf 1992]. This can be structured as a continuation monad, and later work used explicit monadic notation for it [Carette and Kiselyov 2011; Kiselyov et al. 2004; Swadi et al. 2006]. Our definition is as follows:

```
newtype Gen (A : MetaTy) = Gen {unGen : {R : Ty} → (A → ⇑R) → ⇑R}
```

This is a monad in MetaTy in a standard sense:

```
instance Monad Gen where
  return a = Gen $ λ k. k a
  ga >>= f  = Gen $ λ k. unGen ga (λ a. unGen (f a) k))
```

In [Kiselyov et al. 2004],[Carette and Kiselyov 2011] and [Swadi et al. 2006], the answer type is a parameter to the continuation monad, while we have it as polymorphic. We have found that polymorphic answer types are more convenient, because we do not have to anticipate the type of the code output when are defining a code generator. Hence, we can "run" actions as follows:

```
runGen : Gen (⇑A) → ⇑A
runGen ma = unGen ma id
```

From now on, we reuse Haskell-style type classes and **do**-notation in CFTT. We will use type classes in an informal way, without precisely specifying how they work. However, type classes are used in essentially the same way in the Agda and Haskell implementations, with modest technical differences. From the Monad instance, the Functor and Applicative instances can be also derived. We reuse (<$>) and (<*>) for applicative notation as well.

We can let-bind object expressions in Gen:

```
gen : {A : Ty} → ⇑A → Gen (⇑A)
gen a = Gen $ λ k. ⟨let x : A := ~a; ~(k ⟨x⟩)⟩
```

And also recursive definitions of computations:

```
genRec : {A : CompTy} → (⇑A → ⇑A) → Gen (⇑A)
genRec f = Gen $ λ k. ⟨letrec x : A := ~(f ⟨x⟩); ~(k ⟨x⟩)⟩
```

Now, using **do**-notation, we may write **do** {x ← gen ⟨10 + 10⟩; y ← gen ⟨20 + 20⟩}; **return** ⟨x + y⟩}, for a Gen (⇑Nat) action. Running this with runGen yields ⟨**let** x := 10 + 10; **let** y := 20 + 20; x + y⟩. We can also define a "safer" binding-time improvement for products, using let-insertion:

```
up : ⇑(A, B) → Gen (⇑A, ⇑B)
up x = do x ← gen x
          return (⟨fst ~x⟩, ⟨snd ~x⟩)

down : Gen (⇑A, ⇑B) → ⇑(A, B)
down x = do (a, b) ← x
            return ⟨(~a, ~b)⟩
```

3.3 Monads

Let us start with the Maybe monad. We have **data** Maybe A := Nothing | Just A, and Maybe itself is available as a ValTy → ValTy metafunction. However, we cannot directly fashion a monad out of Maybe, since we do not have enough type formers in ValTy. We could try to use the following type for binding:

$$(\gg\!=) : ⇑(\text{Maybe A}) \to (⇑A \to ⇑(\text{Maybe B})) \to ⇑(\text{Maybe B})$$

This works, but the definition necessarily uses runtime case splits on Maybe values, many of which could be optimized away during staging. Also, not having a "real" monad is inconvenient for the purpose of code reuse.

Instead, our strategy is to only use proper monads in MetaTy, and convert between object types and meta-monads when necessary, as a form of binding-time improvement. We define a class for this conversion:

```
class MonadGen M ⇒ Improve (F : ValTy → Ty) (M : MetaTy → MetaTy) where
  up   : {A : ValTy} → ⇑(F A) → M (⇑A)
  down : {A : ValTy} → M (⇓A) → ⇑(F A)
```

Assume that Maybe_M is the standard meta-level monad, and MaybeT_M is the standard monad transformer, defined as follows:

```
newtype MaybeT_M M A = MaybeT_M {runMaybeT_M : M (Maybe_M A)}
```

Now, the binding-time improvement of Maybe is as follows:

```
instance Improve Maybe (MaybeT_M Gen) where
  up ma = MaybeT_M $ Gen $ λ k.
    ⟨case ~ma of Nothing → ~(k Nothing_M); Just a → ~(k (Just_M ⟨a⟩))⟩
  down (MaybeT_M (Gen ma)) =
    ma (λ x. case x of Nothing_M → ⟨Nothing⟩; Just_M a → ⟨Just ~a⟩)
```

With this, we get the Monad instance for free from MaybeT_M and Gen. A small example:

```
let n : Maybe Nat := ~(down $ do {x ← return ⟨10⟩; y ← return ⟨20⟩; return ⟨x + y⟩}); ...
```

Since MaybeT_M is meta-level, its monadic binding fully computes at staging time. Thus, the above code is staged to

```
let n : Maybe Nat := Just (10 + 20); ...
```

Assume also a lift : Monad M ⇒ M A → MaybeT_M M A operation which comes from MaybeT_M being a monad transformer. We can do let-insertion in MaybeT_M Gen by simply lifting:

```
gen' : ⇑A → MaybeT_M Gen (⇑A)
gen' a = lift (gen a)
```

However, it is more convenient to proceed in the style of Haskell's monad transformer library `mtl` [mtl developers 2024], and have a class for monads that support code generation:

```
class Monad M ⇒ MonadGen M where
  liftGen : Gen A → M A
instance MonadGen Gen where liftGen = id
instance MonadGen M ⇒ MonadGen (MaybeT_M M) where liftGen = lift ∘ liftGen
```


Working in Gen is convenient, since we can freely generate object code and also have access to the full metalanguage. Also, the point of staging is that *eventually* all metaprograms will be used for the purpose of code generation, so we eventually runGen all of our actions. So why not just always work in Gen? The implicit nature of Gen may make it harder to reason about the size and content of generated code. This is a bit similar to the IO monad in Haskell [Jones 2001], where eventually everything needs to run in IO, but we may not want to write all of our code in IO.

3.3 Monads

First, following the style of Haskell's monad transformer library `mtl` [mtl developers 2024], we define a class for monads that support code generation.

```
class Monad M ⇒ MonadGen M where
  liftGen : Gen A → M A
```

We also redefine gen and genRec to work in any MonadGen, so from now on we have:

```
gen    : MonadGen M ⇒ ⇑A → M (⇑A)
genRec : MonadGen M ⇒ (⇑A → ⇑A) → M (⇑A)
```

In the rest of this paper, we shall only mention ReaderT, MaybeT and StateT as monad transformers. For all of these, we can define the MonadGen instance simply by liftGen = lift ∘ liftGen.

Let us look at the Maybe monad now. We have **data** Maybe A := Nothing | Just A, and Maybe itself is available as a ValTy → ValTy metafunction. However, we cannot directly fashion a monad out of Maybe, since we do not have polymorphism in ValTy, nor can we store functions inside Maybe. We could try to use the following type for binding:

$$(\gg\!=) : ⇑(\text{Maybe A}) \to (⇑A \to ⇑(\text{Maybe B})) \to ⇑(\text{Maybe B})$$

This would work, but the definition would necessarily use runtime case splits on Maybe values, many of which could be optimized away during staging. Also, not having a "real" monad is inconvenient for the purpose of code reuse.

Instead, our strategy is to only use proper monads in MetaTy, and convert between object types and meta-monads when necessary, as a form of binding-time improvement. We define a class for this conversion:

```
class MonadGen M ⇒ Improve (F : ValTy → Ty) (M : MetaTy → MetaTy) where
  up   : {A : ValTy} → ⇑(F A) → M (⇑A)
  down : {A : ValTy} → M (⇓A) → ⇑(F A)
```

Assume that Maybe_M is the standard meta-level monad, and MaybeT_M is the standard monad transformer, defined as follows:

```
newtype MaybeT_M M A = MaybeT_M {runMaybeT_M : M (Maybe_M A)}
```

Now, the binding-time improvement of Maybe is as follows:

```
instance Improve Maybe (MaybeT_M Gen) where
  up ma = MaybeT_M $ Gen $ λ k.
    ⟨case ~ma of Nothing → ~(k Nothing_M); Just a → ~(k (Just_M ⟨a⟩))⟩
  down (MaybeT_M (Gen ma)) =
    ma (λ x. case x of Nothing_M → ⟨Nothing⟩; Just_M a → ⟨Just ~a⟩)
```

The MonadGen instance can be defined uniformly for every monad transformer as liftGen = lift ∘ liftGen. We also redefine gen and genRec to work in any MonadGen, so from now on we have:

```
gen    : MonadGen M ⇒ ⇑A → M (⇑A)
genRec : MonadGen M ⇒ (⇑A → ⇑A) → M (⇑A)
```

3.3.1 Case splitting in monads. We often want to case-split on object-level data inside a monadic action, and perform different actions in different branches. At first, this may seem problematic, because we cannot directly compute metaprograms from object-level case splits. Fortunately, with a little bit more work, this is actually possible in any MonadGen, for any value type.

We demonstrate this for lists. An object-level **case** on lists introduces two points where code generation can continue. We define a metatype which gives us a "view" on these points:

```
data SplitList A = Nil' | Cons' (⇑A) (⇑(List A))
```

We can generate code for a case split, returning a view on it:

```
split : ⇑(List A) → Gen (SplitList A)
split as = Gen $ λ k. ⟨case ~as of Nil → ~(k Nil'); Cons a as → ~(k (Cons' ⟨a⟩ ⟨as⟩))⟩
```

Now, in any MonadGen, assuming as : ⇑(List A), we may write

```
do {sp ← liftGen (split as); (case sp of Nil' → ...; Cons' a as → ...)}
```

This can be generalized to splitting on any object value. In the Agda and Haskell implementations, we overload split with a class similar to the following:

```
class Split (A : ValTy) where
  SplitTo : MetaTy
  split   : ⇑A → Gen SplitTo
```

In a native implementation of CFTT it may make sense to extend do-notation, so that we elaborate **case** on object values to an application of the appropriate split function. We adopt this in the rest of the paper, so when working in a MonadGen, we can write **case** as **of** Nil → ...; Cons a as → ..., binding a : ⇑A and as : ⇑(List A) in the Cons case.

3.4 Monad Transformers

At this point, it makes sense to aim for a monad transformer library where binding-time improvement is defined compositionally, by recursion on the transformer stack. The base case is the following:

```
newtype Identity A := Identity {runIdentity : A}
instance Improve Identity Gen where
  up x   = return ⟨runIdentity ~x⟩
  down x = unGen x $ λ a. ⟨Identity ~a⟩
```

We recover the object-level Maybe as MaybeT Identity, from the following MaybeT:

```
newtype MaybeT (M : ValTy → Ty) (A : ValTy) := MaybeT {runMaybeT : M (Maybe A)}
```


With this, we get the Monad instance for free from MaybeT_M and Gen. A small example:

```
let n : Maybe Nat := ~(down $ do {x ← return ⟨10⟩; y ← return ⟨20⟩; return ⟨x + y⟩}); ...
```

Since MaybeT_M is meta-level, its monadic binding fully computes at unstaging time. Thus, the above code is computed to

```
let n : Maybe Nat := Just (10 + 20); ...
```

We can also use gen for let-insertion in MaybeT Gen, since it has a MonadGen instance.

3.3.1 Case splitting in monads. We often want to case-split on object-level data inside a monadic action, and perform different actions in different branches. At first, this may seem problematic, because we cannot directly compute metaprograms from object-level case splits. Fortunately, with a little bit more work, this is possible in any MonadGen, for any value type, using a variation of the so-called "trick" [Danvy et al. 1996].

We demonstrate this for lists. An object-level **case** on lists introduces two points where code generation can continue. We define a metatype which gives us a "view" on these points:

```
data SplitList A = Nil' | Cons' (⇑A) (⇑(List A))
```

We can generate code for a case split, returning a view on it:

```
split : ⇑(List A) → Gen (SplitList A)
split as = Gen $ λ k. ⟨case ~as of Nil → ~(k Nil'); Cons a as → ~(k (Cons' ⟨a⟩ ⟨as⟩))⟩
```

Now, in any MonadGen, assuming as : ⇑(List A), we may write

```
do {sp ← liftGen (split as); (case sp of Nil' → ...; Cons' a as → ...)}
```

This can be generalized to splitting on any object value. In the Agda and Haskell implementations, we overload split with a class similar to the following:

```
class Split (A : ValTy) where
  SplitTo : MetaTy
  split   : ⇑A → Gen SplitTo
```

In a native implementation of CFTT it may make sense to extend do-notation, so that we elaborate **case** on object values to an application of the appropriate splitting function. We adopt this in the rest of the paper, so when working in a MonadGen, we can write **case** as **of** Nil → ...; Cons a as → ..., binding a : ⇑A and as : ⇑(List A) in the Cons case.

3.4 Monad Transformers

At this point, it makes sense to aim for a monad transformer library where binding-time improvement is defined compositionally, by recursion on the transformer stack. The base case is the following:

```
newtype Identity A := Identity {runIdentity : A}
instance Improve Identity Gen where
  up x   = return ⟨runIdentity ~x⟩
  down x = unGen x $ λ a. ⟨Identity ~a⟩
```

We recover the object-level Maybe as MaybeT Identity, from the following MaybeT:

```
newtype MaybeT (M : ValTy → Ty) (A : ValTy) := MaybeT {runMaybeT : M (Maybe A)}
```

With this, improvement can be generally defined for MaybeT:

```
instance Improve F M ⇒ Improve (MaybeT F) (MaybeT_M M) where
  up x = MaybeT_M $ do
    ma ← up ⟨runMaybeT ~x⟩
    case ma of Nothing → return Nothing_M
               Just a  → return (Just_M a)
  down (MaybeT_M x) = ⟨MaybeT ~(down $ x ⤜ λ case
    Nothing_M → return ⟨Nothing⟩
    Just_M a  → return ⟨Just ~a⟩)⟩
```

In the **case** in **up**, we use our syntax sugar for matching on a Maybe value inside an M action. This is legal, since we know from the Improve F M assumption that M is a MonadGen. In **down** we also use λ **case** ... to shorten λ x. **case** x **of** ....

In the meta level, we can reuse essentially all definitions from `mtl`. From there, only the continuation monad transformer fails to support binding-time-improvement in CFTT, because of the obvious need for dynamic closures. In the following we only present StateT and ReaderT.

We start with StateT. We assume StateT_M as the standard meta-level definition. The object-level StateT has type (S : ValTy)(F : ValTy → Ty)(A : ValTy) → ValTy; the state parameter S has to be a value type, since it is an input to an object-level function.

```
instance Improve F M ⇒ Improve (StateT S F) (StateT_M (⇑S) M) where
  up x = StateT_M $ λ s. do
    as ← up ⟨runStateT ~x ~s⟩
    case as of (a, s) → return (a, s)
  down x = ⟨StateT (λ s. ~(down $ do
    (a, s) ← runStateT_M x ⟨s⟩
    return ⟨(~a, ~s)⟩))⟩
```

Like before in MaybeT, we rely on object-level case splitting in the definition of **up**. For Reader, the environment parameter also has to be a value type, and we define improvement as follows.

```
instance Improve F M ⇒ Improve (ReaderT R F) (ReaderT_M (⇑R) M) where
  up x   = ReaderT_M $ λ r. up ⟨runReaderT ~x ~r⟩
  down x = ⟨ReaderT (λ r. ~(down (runReaderT_M x ⟨r⟩)))⟩
```

3.4.1 State *and* Reader *operations.* If we use the modify function that we already have in State_M, a curious thing happens. The meaning of modify (λ x. ⟨~x + ~x⟩) is to replace the current state x, as an object expression, with the expression ⟨~x + ~x⟩, and this happens at staging time. This behaves as an "inline" modification which replaces every subsequent mention of the state with a different expression. For instance, ignoring newtype wrappers for now,

```
down $ do {modify (λ x. ⟨~x + ~x⟩); modify (λ x. ⟨~x + ~x⟩); return ⟨()⟩}
```

is staged to

```
⟨λ x. ((), (x + x) + (x + x))⟩
```

which duplicates the evaluation of x + x. The solution is to force the evaluation of the new state in the object language, by let-insertion. A similar phenomenon happens with the local function

With this, improvement can be generally defined for MaybeT:

```
instance Improve F M ⇒ Improve (MaybeT F) (MaybeT_M M) where
  up x = MaybeT_M $ do
    ma ← up ⟨runMaybeT ~x⟩
    case ma of Nothing → return Nothing_M
               Just a  → return (Just_M a)
  down (MaybeT_M x) = ⟨MaybeT ~(down $ x ⤜ λ case
    Nothing_M → return ⟨Nothing⟩
    Just_M a  → return ⟨Just ~a⟩)⟩
```

In the **case** in **up**, we use our syntax sugar for matching on a Maybe value inside an M action. This is legal, since we know from the Improve F M assumption that M is a MonadGen. In **down** we also use λ **case** ... to shorten λ x. **case** x **of** ....

On the meta level, we can borrow essentially all definitions from `mtl`. Only the continuation monad transformer fails to support binding-time-improvement in CFTT, because of the obvious need for dynamic closures. In the following we only present StateT and ReaderT.

We start with StateT. We assume StateT_M as the standard meta-level definition. The object-level StateT has type (S : ValTy)(F : ValTy → Ty)(A : ValTy) → ValTy; the state parameter S has to be a value type, since it is an input to an object-level function.

```
instance Improve F M ⇒ Improve (StateT S F) (StateT_M (⇑S) M) where
  up x = StateT_M $ λ s. do
    as ← up ⟨runStateT ~x ~s⟩
    case as of (a, s) → return (a, s)
  down x = ⟨StateT (λ s. ~(down $ do
    (a, s) ← runStateT_M x ⟨s⟩
    return ⟨(~a, ~s)⟩))⟩
```

Like before in MaybeT, we rely on object-level case splitting in the definition of **up**. For Reader, the environment parameter also has to be a value type, and we define improvement as follows.

```
instance Improve F M ⇒ Improve (ReaderT R F) (ReaderT_M (⇑R) M) where
  up x   = ReaderT_M $ λ r. up ⟨runReaderT ~x ~r⟩
  down x = ⟨ReaderT (λ r. ~(down (runReaderT_M x ⟨r⟩)))⟩
```

3.4.1 State *and* Reader *operations.* If we use the modify function that we already have in State_M, a curious thing happens. The meaning of modify (λ x. ⟨~x + ~x⟩) is to replace the current state x, as an object expression, with the expression ⟨~x + ~x⟩, and this happens at unstaging time. This behaves as an "inline" modification which replaces every subsequent mention of the state with a different expression. For instance, ignoring newtype wrappers for now,

```
down $ do {modify (λ x. ⟨~x + ~x⟩); modify (λ x. ⟨~x + ~x⟩); return ⟨()⟩}
```

is unstaged to

```
⟨λ x. ((), (x + x) + (x + x))⟩
```

which duplicates the evaluation of x + x. The duplication can be avoided by let-binding the result in the object language. A similar phenomenon happens with the local function in Reader. So we

in Reader. So we define "stricter" versions of these operations. We also return ⇑() from actions instead of () — the former is more convenient, because the **down** operation can be immediately used on it.

```
put' : (MonadState (⇑S) M, MonadGen M) ⇒ ⇑S → M (⇑())
put' s = do {s ← gen s; put s; return ⟨()⟩}

modify' : (MonadState (⇑S) M, MonadGen M) ⇒ (⇑S → ⇑S) → M (⇑())
modify' f = do {s ← get; put' (f s)}

local' : (MonadReader (⇑R) M, MonadGen M) ⇒ (⇑R → ⇑R) → M A → M A
local' f ma = do {r ← ask; r ← gen (f r); local (λ _. r) ma}
```

Now,

```
down $ do {modify' (λ x. ⟨~x + ~x⟩); modify' (λ x. ⟨~x + ~x⟩)}
```

is staged to

```
⟨λ x. let x := x + x; let x := x + x; ((), x)⟩
```

3.5 Joining Control Flow in Monads

There is a deficiency in our library so far. Assuming b : ⇑Bool, consider:

```
do (case b of True → put'⟨10⟩; False → put'⟨20⟩)
   modify'(λ x. ⟨~x + 10⟩)
```

The modify'(λ x.⟨~x + 10⟩) action gets inlined in both case branches during staging. This follows from the definition of monadic binding in Gen and the split function in the desugaring of **case**. Code generation is continued in both branches with the same action. If we have multiple Boolean **case** splits sequenced after each other, that yields exponential code size. Right now, we can fix this by let-binding the problematic action:

```
do action ← gen $ down (case b of True → put'⟨10⟩; False → put'⟨20⟩)
   up action
   modify'(λ x. ⟨~x + 10⟩)
```

This removes code duplication by round-tripping through an object-level **let**. This solution is fairly good in a state monad, where **down** only introduces a runtime pair constructor, and it is feasible to compile object-level pairs as unboxed data, without overheads. However, for Maybe, **down** introduces a runtime Just or Nothing, and **up** introduces a runtime case split. A better solution would be to introduce two let-bound *join points* before the offending **case**, one for returning a Just and one for returning Nothing, but fusing away the actual runtime constructors. So, for example, we would like to produce object code which looks like the following:

```
let joinJust n := ...
let joinNothing () := ...
case x == 10 of
  True  → joinJust (x + 10)
  False → joinNothing ()
```


define "stricter" versions of these operations. We also return ⇑() from actions instead of () — the former is more convenient, because the **down** operation can be immediately used on it.

```
put' : (MonadState (⇑S) M, MonadGen M) ⇒ ⇑S → M (⇑())
put' s = do {s ← gen s; put s; return ⟨()⟩}

modify' : (MonadState (⇑S) M, MonadGen M) ⇒ (⇑S → ⇑S) → M (⇑())
modify' f = do {s ← get; put' (f s)}

local' : (MonadReader (⇑R) M, MonadGen M) ⇒ (⇑R → ⇑R) → M A → M A
local' f ma = do {r ← ask; r ← gen (f r); local (λ _. r) ma}
```

Now,

```
down $ do {modify' (λ x. ⟨~x + ~x⟩); modify' (λ x. ⟨~x + ~x⟩)}
```

is unstaged to

```
⟨λ x. let x := x + x; let x := x + x; ((), x)⟩.
```

3.5 Joining Control Flow in Monads

There is a deficiency in our library so far. Consider:

```
f : Bool → StateT Nat (MaybeT Identity) ()
f b := ~(down $ do
  case ⟨b⟩ of True → put'⟨10⟩; False → put'⟨20⟩
  modify' (λ x. ⟨~x + ~x⟩))
```

This is unstaged to the following (omitting newtype wrappers):

```
f b s := case b of
  True  → let s := 10; let s := s + s; Just ((), s)
  False → let s := 20; let s := s + s; Just ((), s)
```

Notice that the final state modification gets inlined into both branches. This duplication follows from the definition of monadic binding in Gen and the split function in the desugaring of **case**. Code generation is continued in both branches with the same action. If we have multiple Boolean **case** splits sequenced after each other, that yields exponential code size. A possible fix is to let-bind the branching action:

```
f b := ~(down $ do
  act ← gen $ down $ case ⟨b⟩ of True → put'⟨10⟩; False → put'⟨20⟩
  up act
  modify (λ x. ⟨~x + ~x⟩))
```

Such fused returns are possible whenever we have a Gen A action at the bottom of the transformer stack, such that A is isomorphic to a meta-level finite sum of value types. Recall that Gen A is defined as {R : ValTy} → (A → ⇑R) → ⇑R. Here, if A is a finite sum, we can rearrange A → ⇑R to a finite product of functions.

We could proceed with finite sums, but we will need finite *sums-of-products* (SOP in short) later in Section in 4, so we develop SOP-s. We view SOP-s as a Tarski-style universe consisting of a type of descriptions and a way of interpreting descriptions into MetaTy ("El" for "elements" of the type).

$$\begin{aligned} &U_{SOP} : \mathsf{MetaTy} && El_{SOP} : U_{SOP} \to \mathsf{MetaTy} \\ &U_{SOP} = \mathsf{List}\ (\mathsf{List}\ \mathsf{ValTy}) && El_{SOP}\ A = \ldots \end{aligned}$$

A type description is a list of lists of value types. We decode this to a sum of products of value types. We omit the definition here. U_{SOP} is closed under value types, finite product types and finite sum types. For instance, we have $\mathsf{Either}_{SOP} : U_{SOP} \to U_{SOP} \to U_{SOP}$ together with $\mathsf{Left}_{SOP} : El_{SOP}\ A \to El_{SOP}\ (\mathsf{Either}_{SOP}\ A\ B)$, $\mathsf{Right}_{SOP} : El_{SOP}\ B \to El_{SOP}\ (\mathsf{Either}_{SOP}\ A\ B)$ and a case splitting operation. It is more convenient to work with type formers in MetaTy, and only convert to SOP representations when needed, so we define a class for the representable types:

```
class IsSOP (A : MetaTy) where
  Rep : U_SOP
  rep : A ≃ El_SOP Rep
```

Above, A ≃ Rep denotes a record containing a pair of back-and-forth functions, together with proofs (as propositional equalities) that they are inverses. We will overload rep as the forward conversion function with type A → El_{SOP} Rep, and write rep^{-1} for its inverse. Now we define an isomorphic presentation of $El_{SOP}\ A \to ⇑R$ as a product of object-level functions:

$$\begin{aligned} &\mathsf{Fun}_{SOP\Uparrow} : U_{SOP} \to \mathsf{Ty} \to \mathsf{MetaTy} \\ &\mathsf{Fun}_{SOP\Uparrow}\ \mathsf{Nil}\ R = () \\ &\mathsf{Fun}_{SOP\Uparrow}\ (\mathsf{Cons}\ A\ B)\ R = (⇑(\mathsf{foldr}\ (\to)\ R\ A),\ \mathsf{Fun}_{SOP\Uparrow}\ B\ R) \end{aligned}$$

$$\begin{aligned} &\mathsf{tabulate} : (El_{SOP}\ A \to ⇑R) \to \mathsf{Fun}_{SOP\Uparrow}\ A\ R \\ &\mathsf{index} : \mathsf{Fun}_{SOP\Uparrow}\ A\ R \to (El_{SOP}\ A \to ⇑R) \end{aligned}$$

We omit here the definitions of tabulate and index. We will also need to let-bind all functions in a $\mathsf{Fun}_{SOP⇑}$:

$$\begin{aligned} &\mathsf{genFun}_{SOP\Uparrow} : \{A : U_{SOP}\} \to \mathsf{Fun}_{SOP\Uparrow}\ A\ R \to \mathsf{Fun}_{SOP⇑}\ A\ R \\ &\mathsf{genFun}_{SOP\Uparrow}\ \{\mathsf{Nil}\}\ () = \mathbf{return}\ () \\ &\mathsf{genFun}_{SOP\Uparrow}\ \{\mathsf{Cons}\ \_\ A\}\ (f,\ fs) = (,) <\$> \mathsf{gen}\ f <*> \mathsf{genFun}_{SOP\Uparrow}\ \{A\}\ fs \end{aligned}$$

We introduce a class for monads that support control flow joining.

```
class Monad M ⇒ MonadJoin M where
  join : IsSOP A ⇒ M A → M A
```


However, this yields suboptimal code:

```
f b s :=
  let act s := case b of True  → let s := 10; Just((), s)
                        False → let s := 20; Just((), s);
  case act s of
    Just (_, s) → let s := s + s; Just ((), s)
    Nothing     → Nothing
```

This solution is fairly good when we only have State or Reader effects, where **down** only introduces a runtime pair constructor, and it is feasible to compile object-level pairs as unboxed data, without overheads. However, for Maybe, **down** introduces a runtime Just or Nothing, and **up** introduces a runtime case split. A better solution would be to introduce two let-bound join points before the offending **case**, one for returning a Just and one for returning Nothing, but fusing away the actual runtime constructors:

```
f b s :=
  let joinNothing s := Nothing;
  let joinJust    s := let s := s + s; Just ((), s);
  case b of
    True  → let s := 10; joinJust s
    False → let s := 20; joinJust s
```

Here, joinNothing happens to be dead code, but it is easy to clean that up in downstream compilation. Such "fused" returns are possible whenever we have a Gen A action at the bottom of the transformer stack, such that A is isomorphic to a meta-level finite sum of value types. Recall that Gen A is defined as {R : ValTy} → (A → ⇑R) → ⇑R. Here, if A is a finite sum, we can rearrange A → ⇑R to a finite product of functions.

We could proceed with finite sums, but we will need finite *sums-of-products* (SOP in short) later in Section in 4, so we develop SOP-s. SOPs have been used in generic and staged programming; see e.g. [de Vries and Löh 2014a] and [Pickering et al. 2020]. We view SOP-s as a Tarski-style universe consisting of a type of descriptions and a way of interpreting descriptions into MetaTy ("El" for "elements" of the type).

$$\begin{aligned} &U_{SOP} : \mathsf{MetaTy} && El_{SOP} : U_{SOP} \to \mathsf{MetaTy} \\ &U_{SOP} = \mathsf{List}\ (\mathsf{List}\ \mathsf{ValTy}) && El_{SOP}\ A = \ldots \end{aligned}$$

A type description is a list of lists of value types. We decode this to a sum of products of value types. For example, El_{SOP} [[Bool, Bool], []] (using Haskell-like list notation) is isomorphic to Either (⇑Bool, ⇑Bool) (). U_{SOP} is closed under value types, finite product types and finite sum types. For instance, we have $\mathsf{Either}_{SOP} : U_{SOP} \to U_{SOP} \to U_{SOP}$ together with $\mathsf{Left}_{SOP} : El_{SOP}\ A \to El_{SOP}\ (\mathsf{Either}_{SOP}\ A\ B)$, $\mathsf{Right}_{SOP} : El_{SOP}\ B \to El_{SOP}\ (\mathsf{Either}_{SOP}\ A\ B)$ and a case splitting operation. It is more convenient to work with type formers in MetaTy, and only convert to SOP representations when needed, so we define a class for the representable types:

```
class IsSOP (A : MetaTy) where
  Rep : U_SOP
  rep : A ≃ El_SOP Rep
```

The most interesting instance is the "base case" for Gen:

```
instance MonadJoin Gen where
  join ma = Gen $ λ k. runGen $ do
    joinPoints ← genFun_SOP⇑ (tabulate (k ∘ rep⁻¹))
    a ← ma
    return $ index joinPoints (rep a)
```

Here we first convert k : A → ⇑R to a product of join points and let-bind each one of them. Then we generate code that returns to the appropriate join point for each return value. Other MonadJoin instances are straightforward. In StateT$_M$, we need the extra IsSOP S constraint because S is returned as a result value.

```
instance (MonadJoin M) ⇒ MonadJoin (MaybeT_M M) where
  join (MaybeT_M ma) = MaybeT_M (join ma)
instance (MonadJoin M) ⇒ MonadJoin (ReaderT_M R M) where
  join (ReaderT_M ma) = ReaderT_M (join ∘ ma)
instance (MonadJoin M, IsSOP S) ⇒ MonadJoin (StateT_M S M) where
  join (StateT_M ma) = StateT_M (join ∘ ma)
```

Now, whenever the return value of a case-splitting action is a sum of products of values (this includes just returning an object value, which is by far the most common situation), we can use join $ case x of ... to eliminate code duplication, without creating runtime sum types.

3.6 Code Example

We look at a slightly larger code example. We define annotated binary trees as

```
data Tree A := Leaf | Node A (Tree A) (Tree A).
```

We write fail : M A for returning Nothing in any monad transformer stack that contains MaybeT. We define a function which traverses a tree and replaces values with values taken from a list. If the tree contains 0, we throw an error. If we run out of list elements, we leave values unchanged.

```
letrec f : Tree Nat → StateT (List Nat) (MaybeT Identity) (Tree Nat)
    f t := ~(down $ case t of
      Leaf → return ⟨Leaf⟩
      Node n l r → do
        case ⟨~n == 0⟩ of True  → fail
                          False → return ()
        ns ← get
        n  ← join $ case ns of Nil         → return n
                               Cons n ns → do {put ns; return n}
        l ← up ⟨f ~l⟩
        r ← up ⟨f ~r⟩
        return ⟨Node ~n ~l ~r⟩)
```


Above, A ≃ Rep denotes a record containing a pair of back-and-forth functions, together with proofs (as propositional equalities) that they are inverses. We will overload rep as the forward conversion function with type A → El$_{SOP}$ Rep, and write rep^{-1} for its inverse.

In 2LTT literature, a metatype A is called *cofibrant* if A → B is isomorphic to an object type whenever B is isomorphic to an object type [Annenkov et al. 2019]. We conjecture that our IsSOP types are exactly the cofibrant types in this sense. However, "cofibrant" is not very descriptive in our setting, so we shall keep writing IsSOP.

Accordingly, we define an isomorphic presentation of El$_{SOP}$ A → ⇑R as a product of object-level functions:

```
Fun_SOP⇑ : U_SOP → Ty → MetaTy
Fun_SOP⇑ Nil          R = ()
Fun_SOP⇑ (Cons A B) R = (⇑(foldr (→) R A), Fun_SOP⇑ B R)

tabulate : (El_SOP A → ⇑R) → Fun_SOP⇑ A R
index    : Fun_SOP⇑ A R → (El_SOP A → ⇑R)
```

We omit here the definitions of tabulate and index. We will also need to let-bind all functions in a Fun$_{SOP⇑}$:

```
genFun_SOP⇑ : {A : U_SOP} → Fun_SOP⇑ A R → Fun_SOP⇑ A R
genFun_SOP⇑ {Nil}       ()       = return ()
genFun_SOP⇑ {Cons _ A} (f, fs) = (,) <$> gen f <*> genFun_SOP⇑ {A} fs
```

We introduce a class for monads that support control flow joining.

```
class Monad M ⇒ MonadJoin M where
  join : IsSOP A ⇒ M A → M A
```

The most interesting instance is the "base case" for Gen:

```
instance MonadJoin Gen where
  join ma = Gen $ λ k. runGen $ do
    joinPoints ← genFun_SOP⇑ (tabulate (k ∘ rep⁻¹))
    a ← ma
    return $ index joinPoints (rep a)
```

Here we first convert k : A → ⇑R to a product of join points and let-bind each one of them. Then we generate code that returns to the appropriate join point for each return value. Other MonadJoin instances are straightforward. In StateT$_M$, we need the extra IsSOP S constraint because S is returned as a result value.

```
instance (MonadJoin M) ⇒ MonadJoin (MaybeT_M M) where
  join (MaybeT_M ma) = MaybeT_M (join ma)
instance (MonadJoin M) ⇒ MonadJoin (ReaderT_M R M) where
  join (ReaderT_M ma) = ReaderT_M (join ∘ ma)
instance (MonadJoin M, IsSOP S) ⇒ MonadJoin (StateT_M S M) where
  join (StateT_M ma) = StateT_M (join ∘ ma)
```

Here, all of the **case** matches are done on object-level values, so they are all desugared to applications of split.

- Note the join: without it, the recursive Node traversal code would be inlined in both case branches.
- We do not use join when checking $\langle \sim n == 0 \rangle$. Here, it happens to be superfluous, since the fail "destroys" all subsequent code generation in the branch, by short-circuiting at the meta-level.
- Also, we can use put instead of the "stricter" put′ because the state modification immediately gets sequenced by the enclosing join point.
- To make object-level recursive calls, we just need to wrap them in **up**.

Here, we intentionally tuned the definition for a nice-looking staging output. However, we could also just default to "safe" choices everywhere: we could use a joint point in every **case** with two or more branches, and use the strict state modifications everywhere. This would only add a small amount of administrative noise that can be easily cleaned up by the downstream compiler.

Let us look at the staging output. We omit newtype wrappers and use nested pattern matching on pairs, but otherwise this is exactly the code that we get in our Agda implementation.

```
letrec f : Tree Nat → StateT (List Nat) (MaybeT Identity) (Tree Nat)
    f := λ t ns. case t of
      Leaf → Just (Leaf, ns)
      Node n l r → case (n == 0) of
        True  → Nothing
        False →
          let joinNothing := λ _. Nothing;
          let joinJust := λ n ns. case f l ns of
                Nothing     → Nothing
                Just (l, ns) → case f r ns of
                  Nothing     → Nothing
                  Just (r, ns) → Just (Node n l r, ns);
          case ns of
            Nil         → joinJust n ns
            Cons n ns → joinJust n ns
```

The only flaw in this code is the joinNothing, which is never called, becase we joined an action that never throws an error. But this is also very easy to clean up after staging.

3.7 Discussion

So far, we have a monad transformer library with the following features:

- Almost all definitions from the well-known Haskell ecosystem of monads and monad transformers can be directly reused, in the meta level.
- We can pattern match on object-level values in monadic code, insert object-level **let**-s with gen and avoid code duplication with join.
- In monadic code, object-level data constructors are only ever created by **down**, and matching on object-level data is only created by split and **up**. Monadic operations are fully fused, and all function calls can be compiled to statically known saturated calls.


```
letrec f : Tree Nat → StateT (List Nat) (MaybeT Identity) (Tree Nat)
  f t := ~(down $ case t of
    Leaf → return (Leaf)
    Node n l r → do
      case (~n == 0) of True  → fail
                        False → return ()
      ns ← get
      n  ← join $ case ns of Nil       → return n
                             Cons n ns → do {put ns; return n}
      l ← up (f ~l)
      r ← up (f ~r)
      return (Node ~n ~l ~r))
```

Fig. 1. Monadic source code

Now, the following definition yields the previously seen code output with joinNothing and joinJust.

```
f b := ~(down $ do
  join $ case (b) of True → put′(10); False → put′(20)
  modify′ (λ x. (~x + ~x)))
```

It might make sense to also have a simple "desugaring" rule that inserts a join whenever we have a object-level case split with more than one branch. At worst, this generates some noise and dead code that is easy to remove by conservative code optimization.

3.6 Code Example

We look at a slightly larger code example in Figure 1. We define annotated binary trees as

```
data Tree A = Leaf | Node A (Tree A) (Tree A).
```

We write fail : M A for returning Nothing in any monad transformer stack that contains MaybeT. We define a function which traverses a tree and replaces values with values taken from a list. If the tree contains 0, we throw an error. If we run out of list elements, we leave values unchanged. Here, all of the **case** matches are done on object-level values, so they are all desugared to applications of split.

- Note the join: without it, the recursive Node traversal code would be inlined in both case branches.
- We do not use join when checking $\langle \sim n == 0 \rangle$. Here, it happens to be superfluous, since the fail "destroys" all subsequent code generation in the branch, by short-circuiting at the meta-level.
- Also, we can use put instead of the "stricter" put′ because the state modification immediately gets sequenced by the enclosing join point.
- To make object-level recursive calls, we just need to wrap them in **up**.

Here, we intentionally tuned the definition for a nice-looking staging output. However, we could also just default to "safe" choices everywhere: we could use a joint point in every **case** with two or

As to potential weaknesses, first, the system as described in this section has some syntactic noise and requires extra attention from programmers. We believe that the noise can be mitigated very effectively in a native CFTT implementation. It was demonstrated in a 2LTT implementation in [Kovács 2022] that almost all quotes and splices are unambiguously inferable, if we require that stages of let-definitions are always specified (as we do here). Moreover, **up** and **down** should be also effectively inferable, using bidirectional elaboration. With such inference, monadic code in CFTT would look only modestly more complicated than in Haskell.

Second, in CFTT we cannot store computations (e.g. functions or State actions) in runtime data structures, nor can we have computations in State state or in Reader environments. However, it would be possible to extend CFTT with a closure type former that converts computations to values, in which case there is no such limitation anymore. Here, closure-freedom would be still available; we would be able to pick where to use or avoid the closure type former.

3.8 Agda & Haskell Implementations

We implemented everything in this section in both Agda and typed Template Haskell. We summarize features and differences:

- The Haskell implementation can be used to generate code that can be further compiled by GHC; here the object language is taken to be Haskell itself. Since Haskell does not distinguish value and computation types, we do not track them in the library, and we do not get guaranteed closure-freedom from GHC.
- In Agda, we postulate all types and terms of the object theory in a faithful way (i.e. equivalently to the CFTT syntax presented here), and take Agda itself to be the metalanguage. Here, we can test "staging" by running Agda programs which compute object expressions. However, we can only inspect staging output and cannot compile or run object programs.
- For sums-of-products in Haskell, we make heavy use of *singleton types* [Eisenberg and Weirich 2012] to emulate dependent types. This adds significant noise. Also, in IsSOP instances we can only define the conversion functions and cannot prove that they are inverses, because Haskell does not have enough support for dependent types.

4 STREAM FUSION

Stream fusion refers to a collection of techniques for generating efficient code from declarative definitions involving streams of values, where intermediate data structures are eliminated. Stream fusion can be broadly grouped into *push* fusion, which is based on Church-encodings of inductive lists, and *pull* fusion, which is based on Church-encodings of coinductive lists; see e.g. [Hinze et al. 2010]. The two styles have different trade-offs, and in practical programming it is a good idea to support both, but in this section we focus on pull streams.

The reason is that pull streams have been historically more difficult to efficiently compile, and we can demonstrate significant improvements in CFTT. We also use dependent types in a more essential way than in the previous section.

4.1 Streams

A pull stream is a meta-level specification of a state machine:

```
data Step S A = Stop | Skip S | Yield A S
data Pull (A : MetaTy) : MetaTy where
  Pull : (S : MetaTy) → IsSOP S ⇒ Gen S → (S → Gen (Step S A)) → Pull A
```



```
letrec f : Tree Nat → StateT (List Nat) (MaybeT Identity) (Tree Nat)
    f := λ t ns. case t of
      Leaf → Just (Leaf, ns)
      Node n l r → case (n == 0) of
        True  → Nothing
        False →
          let joinNothing := λ _. Nothing;
          let joinJust := λ n ns. case f l ns of
                Nothing     → Nothing
                Just (l, ns) → case f r ns of
                  Nothing     → Nothing
                  Just (r, ns) → Just (Node n l r, ns);
          case ns of
            Nil        → joinJust n ns
            Cons n ns → joinJust n ns
```

Fig. 2. Unstaged monadic code

more branches, and use the strict state modifications everywhere, and leave it up to the downstream compiler to clean up the code.

We show the unstaging output on Figure 2. We omit newtype wrappers and use nested pattern matching on pairs, but otherwise this is exactly the code that we get in our Agda implementation. Again, the only flaw in this code is the dead binding for joinNothing.

3.7 Discussion

So far, we have a monad transformer library with the following features:

- Almost all definitions from the well-known Haskell ecosystem of monads and monad transformers can be directly reused, in the meta level.
- We can pattern match on object-level values in monadic code, insert object-level **let**-s with gen and avoid code duplication with join.
- In monadic code, object-level data constructors are only ever created by **down**, and matching on object-level data is only created by split and **up**. Monadic operations are fully fused, and all function calls can be compiled to statically known saturated calls.

As to potential weaknesses, first, the system as described in this section has some syntactic noise and requires extra attention from programmers. We believe that the noise can be mitigated very effectively in a native CFTT implementation. It was demonstrated in a 2LTT implementation in [Kovács 2022] that almost all quotes and splices are unambiguously inferable, if we require that stages of let-definitions are always specified (as we do here). Moreover, **up** and **down** should be also effectively inferable, using bidirectional elaboration. With such inference, monadic code in CFTT would look only slightly more complicated than in Haskell.

Second, in CFTT we cannot store computations (e.g. functions or State actions) in runtime data structures, nor can we have computations in State state or in Reader environments. However, it

In the Pull constructor, S is the type of the internal state which is required to be a sum-of-products of value types by the IsSumVS S constraint. Borrowing terminology from Coutts [Coutts 2011], we call each product in the state a *state shape*.

The next field with type Gen S is the initial state, while transitions are represented by the S → Gen (Step S A) field. Possible transitions are stopping (Stop), transitioning to a new state while outputting a value (Yield) and making a silent transition to a new state (Skip).

Let us see some operations on streams now. Pull is evidently a Functor. It is not quite a "zippy" applicative functor, because its application operator requires an extra IsSOP constraint:

```
repeat : A → Pull A
repeat a = Pull () (return ()) (λ _. return $ Yield a ())

(<*>Pull) : IsSOP A ⇒ Pull (A → B) → Pull A → Pull B
(<*>Pull) (Pull S seed step) (Pull S′ seed′ step′) =
  Pull (S, S′, Maybe A) ((,) <$> seed<*>(, Nothing) <$> seed′) $ λ case
    (s, s′, Just a) → step s >>= λ case
      Stop      → return Stop
      Skip s    → return $ Skip (s, s′, Just a)
      Yield f s → return $ Yield (f a) (s, s′, Nothing)
    (s, s′, Nothing) → step′ s′ >>= λ case
      Stop      → return Stop
      Skip s′   → return $ Skip (s, s′, Nothing)
      Yield a s′ → return $ Skip (s, s′, Just a)
```

In repeat, the state is the unit type, while in (<*>Pull) we take the product of states, and also add another Maybe A that is used to buffer a single A value while we are stepping the Pull (A → B) stream. The IsSOP constraints for the new machine states are implicitly dispatched by instance resolution; this works in the Agda and Haskell versions too. We can derive zip and zipWith from (<*>Pull) and (<$>) for streams, with the restriction that the zipped streams must all produce IsSOP values.

Pull is also a monoid with stream appending as the binary operation.

```
empty : Pull A
empty = Pull () (return ()) (λ _. return Stop)

(<>) : Pull A → Pull A → Pull A
(<>) (Pull S seed step) (Pull S′ seed′ step′) = Pull (Either S S′) (Left <$> seed) λ case
  Left s  → step s >>= λ case  Stop    → (Skip ∘ Right) <$> seed′
                               Skip s  → return $ Skip (Left s)
                               Yield a s → return $ Yield a (Left s)
  Right s′ → step s′ >>= λ case Stop    → return Stop
                               Skip s′  → return $ Skip (Right s′)
                               Yield a s′ → return $ Yield a (Right s′)
```


would be possible to extend CFTT with a closure type former that converts computations to values, in which case there is no such limitation anymore. Here, closure-freedom would be still available; we would be able to pick where to use or avoid the closure type former.

3.8 Agda & Haskell Implementations

We implemented everything in this section in both Agda and typed Template Haskell. We summarize features and differences:

- The Haskell implementation can be used to generate code that can be further compiled by GHC; here the object language is taken to be Haskell itself. Since Haskell does not distinguish value and computation types, we do not track them in the library, and we do not get guaranteed closure-freedom from GHC.
- In Agda, we postulate all types and terms of the object theory in a faithful way (i.e. equivalently to the CFTT syntax presented here), and take Agda itself to be the metalanguage. Here, we can test "staging" by running Agda programs which compute object expressions. However, we can only inspect staging output and cannot compile or run object programs.
- For sums-of-products in Haskell, we make heavy use of *singleton types* [Eisenberg and Weirich 2012] to emulate dependent types. This adds significant noise. Also, in IsSOP instances we can only define the conversion functions and cannot prove that they are inverses, because Haskell does not have enough support for dependent types.

4 STREAM FUSION

Stream fusion refers to a collection of techniques for generating efficient code from declarative definitions involving streams of values, where intermediate data structures are eliminated. Stream fusion can be broadly grouped into *push* fusion, which is based on Church-encodings of inductive lists, and *pull* fusion, which is based on Church-encodings of coinductive lists; see e.g. [Hinze et al. 2010]. The two styles have different trade-offs, and in practical programming it is a good idea to support both, but in this section we focus on pull streams.

The reason is that pull streams have been historically more difficult to efficiently compile, and we can demonstrate significant improvements in CFTT. We also use dependent types in a more essential way than in the previous section.

4.1 Streams

A pull stream is a meta-level specification of a state machine:

```
data Step S A = Stop | Skip S | Yield A S
data Pull (A : MetaTy) : MetaTy where
  Pull : (S : MetaTy) → IsSOP S ⇒ Gen S → (S → Gen (Step S A)) → Pull A
```

In the Pull constructor, S is the type of the internal state which is required to be a sum-of-products of value types by the IsSOP S constraint. The next field with type Gen S is the initial state, while transitions are represented by the S → Gen (Step S A) field. Possible transitions are stopping (Stop), transitioning to a new state while outputting a value (Yield) and making a silent transition to a new state (Skip).

Our definition roughly follows the non-staged stream fusion setup of Coutts [Coutts 2011]; the difference is the addition of IsSOP and Gen in our version. Also, borrowing terminology from Coutts, we call each product in the state a *state shape*.

These definitions are standard for streams; compared to the unstaged definitions in previous literature, the only additional noise is just the Gen monad in the initial states and the transitions. Likewise, we can give standard definitions for usual stream functions such as filter, take or drop.

4.2 Running Streams with Mutual Recursion

How do we generate object code from streams? The state S is given as a finite sums of products, but the sums and the products are on the meta level, so we cannot directly use S in object code. Similarly as in the the treatment of join points, we tabulate the $S \to \mathsf{Gen}\ \mathsf{Step}\ S\ A$ transition function to a product of functions. However, these functions need to be *mutually recursive*, since it is possible to transition from any state to any other state, and each such transition is represented as a function call. This problem of generating well-typed mutual blocks was addressed by Yallop and Kiselyov in [Yallop and Kiselyov 2019]. Unlike ibid., which used control effects and mutable references in MetaOCaml, we present a solution that does not use side effects in the metalanguage.

The solution is to extend CompTy with finite products of computations, i.e. assume $() : \mathsf{CompTy}$ and $(,) : \mathsf{CompTy} \to \mathsf{CompTy} \to \mathsf{CompTy}$, together with pairing and projections. These types, like functions, are call-by-name in the runtime semantics, and they also cannot escape the scope of their definition. Hence, we can also "saturate" programs that involve computational product types: every computation definition at type (A, B) can be translated to a pair, and every projection of a let-defined variable can be statically matched up with a pairing in the variable definition. Thus, a recursive let-definition at type $(A \to B, A \to B)$ can be always compiled to a pair of mutual functions.

We redefine the previous $\mathsf{Fun}_{\mathsf{SOP}\Uparrow}$ to return a computation type instead:

```
Fun_SOP⇑ : U_SOP → Ty → CompTy
Fun_SOP⇑ Nil         R = ()
Fun_SOP⇑ (Cons A B) R = (foldr (→) R A, Fun_SOP⇑ B R)

tabulate : (El_SOP A → ⇑R) → ⇑(Fun_SOP⇑ A R)
index    : ⇑(Fun_SOP⇑ A R) → (El_SOP A → ⇑R)
```

Even for join points, this is just as efficient as the previous $\mathsf{Fun}_{\mathsf{SOP}\Uparrow}$ version, since a definition of a product of functions will get compiled to a sequence of function definitions. In addition, we can use it to generate object code from streams. There are several choices for this, but in CFTT the foldr function is as good as we can get.

```
foldr : {A : MetaTy}{B : Ty} → (A → ⇑B → ⇑B) → ⇑B → Pull A → ⇑ B
foldr {A} {B} (Pull S seed step) f b = ⟨
  letrec fs : Fun_SOP⇑ (Rep {S}) B := ~⟨tabulate $ λ s. unGen (step (rep⁻¹ s)) $ λ case
       Stop    → b
       Skip s  → index ⟨fs⟩ (rep s)
       Yield a s → f a (index ⟨fs⟩ (rep s))⟩;
  ~⟨unGen seed $ λ s. index ⟨fs⟩ (rep s)⟩⟩
```

This foldr is quite flexible becuse we can compute any object value from it, including functions. For instance, we can define foldl from foldr:

```
foldl : {A : MetaTy}{B : ValTy} → (⇑B → A → ⇑B) → ⇑B → Pull A → ⇑ B
foldl f b as = ~⟨foldr (λ a g. ⟨λ b. ~g ~⟨f ⟨b⟩ ~⟨a⟩⟩⟩) ⟨λ b. b⟩ as) ~b
```


Let us see some operations on streams now. Pull is evidently a Functor. It is not quite a "zippy" applicative functor, because its application operator requires an extra IsSOP constraint:

```
repeat : A → Pull A
repeat a = Pull () (return ()) (λ _. return $ Yield a ())

(<*>_Pull) : IsSOP A ⇒ Pull (A → B) → Pull A → Pull B
(<*>_Pull) (Pull S seed step) (Pull S' seed' step') =
  Pull (S, S', Maybe A) ((,) <$> seed<*>(, Nothing) <$> seed') $ λ case
    (s, s', Just a) → step s >>= λ case
      Stop      → return Stop
      Skip s    → return $ Skip (s, s', Just a)
      Yield f s → return $ Yield (f a) (s, s', Nothing)
    (s, s', Nothing) → step' s' >>= λ case
      Stop      → return Stop
      Skip s'   → return $ Skip (s, s', Nothing)
      Yield a s' → return $ Skip (s, s', Just a)
```

In repeat, the state is the unit type, while in $(<*>_{\mathsf{Pull}})$ we take the product of states, and also add another Maybe A that is used to buffer a single value while we are stepping the Pull $(A \to B)$ stream. The IsSOP constraints for the new machine states are implicitly dispatched by instance resolution; this works in the Agda and Haskell versions too. We can derive zip and zipWith from $(<*>_{\mathsf{Pull}})$ and (<$>) for streams, with the restriction that the zipped streams must all produce IsSOP values.

Pull is also a monoid with stream appending as the binary operation.

```
empty : Pull A
empty = Pull () (return ()) (λ _. return Stop)

(<>) : Pull A → Pull A → Pull A
(<>) (Pull S seed step) (Pull S' seed' step') = Pull (Either S S') (Left <$> seed) λ case
  Left s  → step s >>= λ case  Stop    → (Skip ∘ Right) <$> seed'
                               Skip s  → return $ Skip (Left s)
                               Yield a s → return $ Yield a (Left s)
  Right s' → step s' >>= λ case Stop    → return Stop
                               Skip s'  → return $ Skip (Right s')
                               Yield a s' → return $ Yield a (Right s')
```

These definitions are standard for streams; compared to the non-staged definitions in previous literature, the only additional noise is just the Gen monad in the initial states and the transitions. Likewise, we can give standard definitions for usual stream functions such as filter, take or drop.

4.2 Running Streams with Mutual Recursion

How do we generate object code from streams? The state S is given as a finite sums of products, but the sums and the products are on the meta level, so we cannot directly use S in object code. Similarly

Note that since we abstract B in a runtime function, it must be a value type. Here, each Stop and Yield in the transition function gets interpreted as a λ-expression in the output. However, those λ-s will be lifted out in the scope, yielding a proper mutually tail-recursive definition with an accummulator for B. Contrast this to the GHC base library, where foldl for lists is also defined from foldr, to enable push-based fusion, but where a substantial *arity analysis* is used in the compiler to (incompletely) eliminate the intermediate closures [Breitner 2014].

4.3 concatMap for Streams

Can we have a list-like Monad instance for streams, with singleton streams for **return** and concatMap for binding? This is not possible. Aiming at

$$\mathsf{concatMap} : (A \to \mathsf{Pull}\,B) \to \mathsf{Pull}\,A \to \mathsf{Pull}\,B,$$

the $A \to \mathsf{Pull}\,B$ function can contain an infinite number of different machine state types, which cannot be represented in a finite amount of object code. Here by "infinite" we mean the notion that is internally available in the meta type theory. For instance, we can define a $\mathsf{Nat}_M \to \mathsf{Pull}\,B$ function which for each $n : \mathsf{Nat}_M$ produces a concatenation of n streams. Hence, we shall have the following function instead:

$$\mathsf{concatMap} : \mathsf{IsSOP}\,A \Rightarrow (A \to \mathsf{Pull}\,B) \to \mathsf{Pull}\,A \to \mathsf{Pull}\,B$$

The idea is the following: if U_{SOP} is closed Σ-types, we can directly define this concatMap, by taking the appropriate dependent sum of the $A \to U_{SOP}$ family of machine states, which we extract from the $A \to \mathsf{Pull}\,B$ function. Let us write $\Sigma\,A\,B : \mathsf{MetaTy}$ for dependent sums, for $A : \mathsf{MetaTy}$ and $B : A \to \mathsf{MetaTy}$, and reuse $(,)$ for pairing. We also use the following field projection functions: $\mathsf{projS} : \mathsf{Pull}\,A \to \mathsf{MetaTy}$, $\mathsf{projSeed} : (as : \mathsf{Pull}\,A) \to \mathsf{projS}\,as$, and $\mathsf{projStep} : (as : \mathsf{Pull}\,A) \to \mathsf{projS}\,as \to \mathsf{Gen}\,(\mathsf{Step}\,(\mathsf{projS}\,as)\,A)$. For now, we assume that the following instance exists:

$$\textbf{instance}\ (\mathsf{IsSOP}\,A,\ \{a : A\} \to \mathsf{IsSOP}\,(B\,a)) \Rightarrow \mathsf{IsSOP}\,(\Sigma\,A\,B)$$

Above, $\{a : A\} \to \mathsf{IsSOP}\,(B\,a)$ is a universally quantified instance constraint; this can be also represented in Agda. The definition of concatMap is as follows.

```
concatMap : IsSOP A ⇒ (A → Pull B) → Pull A → Pull B
concatMap {A} {B} f (Pull S seed step) =
  Pull (S, Maybe (Σ A (projS ∘ f))) ((,) <$> seed <*> return Nothing) $ λ case
    (s, Nothing) → step s >>= λ case
      Stop      → return Stop
      Skip s    → return $ Skip (s, Nothing)
      Yield a s → do {s' ← projSeed (f a); return $ Skip (s, Just (a, s'))}
    (s, Just (a, s')) → projStep (f a) s' >>= λ case
      Stop      → return $ Skip (s, Nothing)
      Skip s'   → return $ Skip (s, Just (a, s'))
      Yield b s' → return $ Yield b (s, Just (a, s'))
```

Here, Nothing marks the states where we are in the "outer" loop, running the Pull A stream until we get its next value. Just marks the states of the "inner" loop, where we have a concrete $a : A$ value and we run the $(f\,a)$ stream until it stops. In the inner loop, the machine state type depends on the $a : A$ value, hence the need for Σ. How do we get IsSOP for Σ? The key observations are:

- Metaprograms cannot inspect the structure of object terms.

as in the the treatment of join points, we tabulate the $S \to \mathsf{Gen}\,\mathsf{Step}\,S\,A$ transition function to a product of functions. However, these functions need to be *mutually recursive*, since it is possible to transition from any state to any other state, and each such transition is represented as a function call. This problem of generating well-typed mutual blocks was addressed by Yallop and Kiselyov in [Yallop and Kiselyov 2019]. In contrast to this work, which used control effects and mutable references in MetaOCaml, we present a solution that does not use side effects in the metalanguage.

The solution is to extend CompTy with finite products of computations, i.e. assume $() : \mathsf{CompTy}$ and $(,) : \mathsf{CompTy} \to \mathsf{CompTy} \to \mathsf{CompTy}$, together with pairing and projections. These types, like functions, are call-by-name in the runtime semantics, and they also cannot escape the scope of their definition. Hence, we can also "saturate" programs that involve computational product types: every computation definition at type (A, B) can be translated to a pair, and every projection of a let-defined variable can be statically matched up with a pairing in the variable definition. Thus, a recursive let-definition at type $(A \to B, A \to B)$ can be always compiled to a pair of mutual functions.

We redefine the previous $\mathsf{Fun}_{\mathsf{SOP}\Uparrow}$ to return a computation type instead:

$$\begin{aligned}
&\mathsf{Fun}_{\mathsf{SOP}\Uparrow} : U_{SOP} \to \mathsf{Ty} \to \mathsf{CompTy}\\
&\mathsf{Fun}_{\mathsf{SOP}\Uparrow}\ \mathsf{Nil}\qquad\quad R = ()\\
&\mathsf{Fun}_{\mathsf{SOP}\Uparrow}\ (\mathsf{Cons}\,A\,B)\,R = (\mathsf{foldr}\,(\to)\,R\,A,\ \mathsf{Fun}_{\mathsf{SOP}\Uparrow}\,B\,R)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{tabulate} : (\mathsf{El}_{SOP}\,A \to \Uparrow R) \to \Uparrow(\mathsf{Fun}_{\mathsf{SOP}\Uparrow}\,A\,R)\\
&\mathsf{index}\quad\ : \Uparrow(\mathsf{Fun}_{\mathsf{SOP}\Uparrow}\,A\,R) \to (\mathsf{El}_{SOP}\,A \to \Uparrow R)
\end{aligned}$$

Even for join points, this is just as efficient as the previous $\mathsf{Fun}_{\mathsf{SOP}\Uparrow}$ version, since a definition of a product of functions will get compiled to a sequence of function definitions. In addition, we can use it to generate object code from streams. There are several choices for this, but in CFTT the foldr function is as good as we can get.

```
foldr : {A : MetaTy}{B : Ty} → (A → ⇑B → ⇑B) → ⇑B → Pull A → ⇑ B
foldr {A} {B} f b (Pull S seed step) = ⟨
  letrec fs : Fun_SOP⇑ (Rep {S}) B := ~(tabulate $ λ s. unGen (step (rep⁻¹ s)) $ λ case
      Stop      → b
      Skip s    → index ⟨fs⟩ (rep s)
      Yield a s → f a (index ⟨fs⟩ (rep s)));
  ~(unGen seed $ λ s. index ⟨fs⟩ (rep s))⟩
```

This foldr is quite flexible because it can produce object terms with computation types. For instance, we can define foldl from foldr:

```
foldl : {A : MetaTy}{B : ValTy} → (⇑B → A → ⇑B) → ⇑B → Pull A → ⇑ B
foldl f b as = ~(foldr (λ a g. ⟨λ b. ~g ~(f ⟨b⟩ ~a)⟩) ⟨λ b. b⟩ as) ~b
```

Note that since we abstract B in a runtime function, it must be a value type. Here, each Stop and Yield in the transition function gets interpreted as a λ-expression in the output. However, those λ-s will be lifted out in the scope, yielding a proper mutually tail-recursive definition with an accummulator for B. Contrast this to the GHC base library, where foldl for lists is also defined from foldr, to enable push-based fusion, but where a substantial *arity analysis* is used in the compiler to (incompletely) eliminate the intermediate closures [Breitner 2014].

- Object types do not depend on object terms.

Hence, we expect that during staging, every $f : \Uparrow A \to \mathrm{ValTy}$ function has to be constant. This is actually true in the staging semantics of CFTT. There, all metafunctions are stable under object-level parallel substitutions. Also, object types are untouched by substitution. Hence, a straightforward unfolding of semantic definitions validates the constancy of f.

Generally speaking, in the semantics, every function whose domain is a product of object types and whose codomain is a constant presheaf, is a constant function. We may call these constancy statements *generativity axioms*, since they reflect the inability of metaprograms to inspect terms, and "generativity" often refers to this property in the staged compilation literature.

Let us write $U_P = \mathrm{List}\,\mathrm{ValTy}$ and $El_P : U_P \to \mathrm{MetaTy}$ for a universe of finite products of value types. Concretely, in CFTT and the Agda implementation, we assume the following:

Axiom (generativity). *For every* $f : El_P\,A \to U_{SOP}$ *and* $a, a' : El_P\,A$, *we have that* $f\,a = f\,a'$.

We remark that there is no risk of staging getting stuck on this axiom, because propositional equality proofs get erased, as we described in Section 2.4.

From generativity, we derive $\Sigma_{SOP} : (A : U_{SOP}) \to (B : El_{SOP}\,A \to U_{SOP}) \to U_{SOP}$ as follows. First, for each $A : U_P$, we define $\mathrm{loop}_A : El_P\,A$ as a product of non-terminating object programs. This is only needed to get arbitrary inhabitants with which we can call $El_P\,A \to U_{SOP}$ functions.

Then, $\Sigma_{SOP}\,A\,B$ is defined as the concatenation of $A_i \times B\,(\mathrm{inject}_i\,\mathrm{loop}_{A_i})$ for each $A_i \in A$, where $(\times)$ is the product type former in U_{SOP} and inject_i has type $El_P\,A_i \to El_{SOP}\,A$. This is similar to the definition of non-dependent products in U_{SOP}, except that we have to get rid of the type dependency by instantiating B with arbitrary programs.

Then, we can show using the generativity axiom that Σ_{SOP} supports projections, pairing and the $\beta\eta$-rules. These are all needed when we define the IsSOP instance, when we have to prove that encoding via rep is an isomorphism. Concretely, assuming IsSOP A and $\{a : A\} \to \mathrm{IsSOP}\,(B\,a)$, we define Rep and rep for $\Sigma\,A\,B$ as follows:

$$\mathrm{Rep}\,\{\Sigma\,A\,B\} = \Sigma\,(\mathrm{Rep}\,\{A\})\,(\lambda x.\,\mathrm{Rep}\,\{B\,(\mathrm{rep}^{-1}\,x)\})$$

$$\mathrm{rep}\,\{\Sigma\,A\,B\}\,(a,\,b) = (\mathrm{rep}\,\{A\}\,a,\ \mathrm{rep}\,\{B\,(\mathrm{rep}^{-1}\,(\mathrm{rep}\,x))\}\,b)$$

Note that rep is only well-typed up to the fact that $\mathrm{rep}^{-1} \circ \mathrm{rep} = \mathrm{id}$; the Agda definition contains an additional transport that we omit here. This is, in fact, our reason for including the isomorphism equations in IsSOP. This, in turn, necessitates using SOP instead of finite sums. We can define a product type former for finite sums of value types, by taking the pairwise products of components. However, we can only take the *object-level* products here, and since object-level products have no $\beta\eta$-rules, we cannot prove $\beta\eta$ for the derived product type in finite sums, and likewise for the derived Σ-type. When we use SOP instead, we do not have to use object-level products and this issue does not appear.

We skip the full definition of Σ_{SOP} here. The reader may refer to the SOP module in the Agda implementation, which is altogether 260 lines with all IsSOP instances.

4.4 Let-Insertion & Case Splitting in Monadic Style

While Pull is not a monad, and hence also not a MonadGen, we can still use a monadic style of stream programming with good ergonomics. First, we need singleton streams for "returning":

```
single : A → Pull A
single a = Pull Bool_M True $ λ b. return $ if b then Yield a False else Stop
```


4.3 concatMap for Streams

Can we have a list-like Monad instance for streams, with singleton streams for **return** and concatMap for binding? This is not possible. Aiming at

$$\mathrm{concatMap} : (A \to \mathrm{Pull}\,B) \to \mathrm{Pull}\,A \to \mathrm{Pull}\,B,$$

the $A \to \mathrm{Pull}\,B$ function can contain an infinite number of different machine state types, which cannot be represented in a finite amount of object code. Here by "infinite" we mean the notion that is internally available in the meta type theory. For instance, we can define a $\mathrm{Nat}_M \to \mathrm{Pull}\,B$ function which for each $n : \mathrm{Nat}_M$ produces a concatenation of n streams. Hence, we shall have the following function instead:

$$\mathrm{concatMap} : \mathrm{IsSOP}\,A \Rightarrow (A \to \mathrm{Pull}\,B) \to \mathrm{Pull}\,A \to \mathrm{Pull}\,B$$

The idea is the following: if U_{SOP} is closed under Σ-types, we can directly define this concatMap, by taking the appropriate dependent sum of the $A \to U_{SOP}$ family of machine states, which we extract from the $A \to \mathrm{Pull}\,B$ function. Let us write $\Sigma\,A\,B : \mathrm{MetaTy}$ for dependent sums, for $A : \mathrm{MetaTy}$ and $B : A \to \mathrm{MetaTy}$, and reuse $(,)$ for pairing. We also use the following field projection functions: $\mathrm{projS} : \mathrm{Pull}\,A \to \mathrm{MetaTy}$, $\mathrm{projSeed} : (as : \mathrm{Pull}\,A) \to \mathrm{projS}\,as$, and $\mathrm{projStep} : (as : \mathrm{Pull}\,A) \to \mathrm{projS}\,as \to \mathrm{Gen}\,(\mathrm{Step}\,(\mathrm{projS}\,as)\,A)$. For now, we assume that the following instance exists:

$$\textbf{instance}\ (\mathrm{IsSOP}\,A,\ \{a : A\} \to \mathrm{IsSOP}\,(B\,a)) \Rightarrow \mathrm{IsSOP}\,(\Sigma\,A\,B)$$

Above, $\{a : A\} \to \mathrm{IsSOP}\,(B\,a)$ is a universally quantified instance constraint; this can be also represented in Agda. The definition of concatMap is as follows.

```
concatMap : IsSOP A ⇒ (A → Pull B) → Pull A → Pull B
concatMap {A} {B} f (Pull S seed step) =
  Pull (S, Maybe (Σ A (projS ∘ f))) (_,_) <$> seed <*> return Nothing) $ λ case
    (s, Nothing) → step s >>= λ case
      Stop      → return Stop
      Skip s    → return $ Skip (s, Nothing)
      Yield a s → do {s' ← projSeed (f a); return $ Skip (s, Just (a, s'))}
    (s, Just (a, s')) → projStep (f a) s' >>= λ case
      Stop      → return $ Skip (s, Nothing)
      Skip s'   → return $ Skip (s, Just (a, s'))
      Yield b s' → return $ Yield b (s, Just (a, s'))
```

Here, Nothing marks the states where we are in the "outer" loop, running the Pull A stream until we get its next value. Just marks the states of the "inner" loop, where we have a concrete $a : A$ value and we run the $(f\,a)$ stream until it stops. In the inner loop, the machine state type depends on the $a : A$ value, hence the need for Σ. How do we get IsSOP for Σ? The key observations are:

- Metaprograms cannot inspect the structure of object terms.
- Object types do not depend on object terms.

Hence, we expect that during staging, every $f : \Uparrow A \to \mathrm{ValTy}$ function has to be constant. This is actually true in the staging semantics of CFTT. There, all metafunctions are stable under object-level parallel substitutions. Also, object types are untouched by substitution. Hence, a straightforward unfolding of semantic definitions validates the constancy of f.

Now, we should use this operation with care, since it has two states and can contribute to a code size blow-up. For example, concatMap over single introduces at least a doubling of the number of state shapes. Although let-insertion and case splitting could be derived as concatMap over single, to avoid the size explosion we give more specialized definitions instead. First, we define a helper function for binding a single IsSOP value.

```
bind_single : IsSOP A' ⇒ ⇑A → (⇑A → Gen A') → (A' → Pull B) → Pull B
bind_single a f g =
  Pull (Σ A' (projS ∘ g)) (do {a' ← f a; s ← projSeed (g a'); return (a', s)}) $ λ case
    (a', s) → projStep (g a') s >>= λ case
      Stop     → return Stop
      Skip s   → return $ Skip (a', s)
      Yield b s → return $ Yield b (a', s)
```

We recover let-insertion and case splitting as follows:

```
gen_Pull : ⇑A → (⇑A → Pull B) → Pull B
gen_Pull a = bind_single a gen

case_Pull : (Split A, IsSOP (SplitTo {A})) ⇒ ⇑A → (SplitTo {A} → Pull B) → Pull B
case_Pull a = bind_single a split
```

Let us look at small example. We write forEach for concatMap with flipped arguments.

```
forEach (take ⟨100⟩ (countFrom ⟨0⟩)) $ λ x.
gen_Pull ⟨~x * 2⟩ $ λ y.
case_Pull ⟨~x < 50⟩ $ λ case
  True  → take y (countFrom x)
  False → single y
```

Here, in every forEach iteration, $\text{gen}_{\text{Pull}}\ \langle \sim x * 2 \rangle$ evaluates the given expression and saves the result to the machine state. Then, $\text{case}_{\text{Pull}}$ branches on an object-level Boolean expression. If we use foldr to generate object code from this definition, we get *four* mutually defined functions. We get two state shapes from single y and one from take y (countFrom x); we sum these to get three for the $\text{case}_{\text{Pull}}$, then finally we get an extra shape from forEach which introduces an additional Maybe.

4.5 Discussion

Our stream library has a strong support for programming in a monadic style, even though Pull is not literally a monad. We can bind object values with concatMap, and we can also do let-insertion and case splitting for them. We also get guaranteed well-typing, closure-freedom, and arbitrary mixing of zipping and concatMap.

We highlight the usage of the generativity axiom as well. Previously in staged compilation, intensional analysis (i.e. the ability to analyze object code) has been viewed as a desirable feature that increases the expressive power of a system. To our knowledge, our work is the first one that exploits the *lack* of intensional analysis in metaprogramming. This is a bit similar to parametricity in type theories, where the inability to analyze types has a payoff in the form of "free theorems" [Wadler 1989].

Generally speaking, in the semantics, every function whose domain is a product of object types and whose codomain is a constant presheaf, is a constant function. We may call these constancy statements *generativity axioms*, since they reflect the inability of metaprograms to inspect terms, and "generativity" often refers to this property in the staged compilation literature.

Let us write $\text{U}_\text{P} = \text{List ValTy}$ and $\text{El}_\text{P} : \text{U}_\text{P} \to \text{MetaTy}$ for a universe of finite products of value types. Concretely, in CFTT and the Agda implementation, we assume the following:

Axiom (generativity). *For every* $\text{f} : \text{El}_\text{P}\ \text{A} \to \text{U}_\text{SOP}$ *and* $\text{a}, \text{a}' : \text{El}_\text{P}\ \text{A}$, *we have that* $\text{f a} = \text{f a}'$.

We remark that there is no risk of staging getting stuck on this axiom, because propositional equality proofs get erased, as we described in Section 2.4.

From generativity, we derive $\Sigma_{\text{SOP}} : (\text{A} : \text{U}_\text{SOP}) \to (\text{B} : \text{El}_\text{SOP}\ \text{A} \to \text{U}_\text{SOP}) \to \text{U}_\text{SOP}$ as follows. First, for each $\text{A} : \text{U}_\text{P}$, we define $\text{loop}_\text{A} : \text{El}_\text{P}\ \text{A}$ as a product of non-terminating object programs. This is only needed to get arbitrary inhabitants with which we can call $\text{El}_\text{P}\ A \to \text{U}_\text{SOP}$ functions.

Then, $\Sigma_{\text{SOP}}\ \text{A B}$ is defined as the concatenation of $\text{A}_i \times \text{B}\ (\text{inject}_i\ \text{loop}_{\text{A}_i})$ for each $\text{A}_i \in \text{A}$, where $(\times)$ is the product type former in U_SOP and inject_i has type $\text{El}_\text{P}\ \text{A}_i \to \text{El}_\text{SOP}\ \text{A}$. This is similar to the definition of non-dependent products in U_SOP, except that we have to get rid of the type dependency by instantiating B with arbitrary programs.

Then, we can show using the generativity axiom that Σ_{SOP} supports projections, pairing and the $\beta\eta$-rules. These are all needed when we define the IsSOP instance, when we have to prove that encoding via rep is an isomorphism. Concretely, assuming IsSOP A and $\{\text{a} : \text{A}\} \to \text{IsSOP}\ (\text{B a})$, we define Rep and rep for $\Sigma\ \text{A B}$ as follows:

$$\text{Rep}\ \{\Sigma\ \text{A B}\} = \Sigma\ (\text{Rep}\ \{\text{A}\})\ (\lambda\ \text{x}.\ \text{Rep}\ \{\text{B}\ (\text{rep}^{-1}\ \text{x})\})$$

$$\text{rep}\ \{\Sigma\ \text{A B}\}\ (\text{a}, \text{b}) = (\text{rep}\ \{\text{A}\}\ \text{a},\ \text{rep}\ \{\text{B}\ (\text{rep}^{-1}\ (\text{rep}\ \text{x}))\}\ \text{b})$$

Note that rep is only well-typed up to the fact that $\text{rep}^{-1} \circ \text{rep} = \text{id}$; the Agda definition contains an additional transport that we omit here. This is, in fact, our reason for including the isomorphism equations in IsSOP.

This, in turn, necessitates using SOP instead of finite sums, for the following reason. We can define a product type former for finite sums of value types, by taking the pairwise products of components, but in this case the products would be object-level products, which do not support $\beta\eta$ rules. This implies that we cannot prove $\beta\eta$-rules for the derived product type in finite sums. In contrast, when we define products for SOP-s, we take *meta-level* pairwise products of components, which does support β and η.

We skip the full definition of Σ_{SOP} here. The reader may refer to the SOP module in the Agda implementation, which is altogether 260 lines with all IsSOP instances.

4.4 Let-Insertion & Case Splitting in Monadic Style

While Pull is not a monad, and hence also not a MonadGen, we can still use a monadic style of stream programming with good ergonomics. First, we need singleton streams for "returning":

```
single : A → Pull A
single a = Pull Bool_M True $ λ b. return $ if b then Yield a False else Stop
```

Now, we should use this operation with care, since it has two states and can contribute to a code size blow-up. For example, concatMap over single introduces at least a doubling of the number of state shapes. Although let-insertion and case splitting could be derived as concatMap over single, to avoid the size explosion we give more specialized definitions instead. First, we define an embedding

Regarding the practical application of our stream library, we think that it would make sense to support both push and pull fusion in a realistic implementation, and allow users to benefit from the strong points of both. Push streams, which we do not present in this paper, have proper Monad and MonadGen instances and are often more convenient to use in CFTT. They are also better for deep traversals of structures where they can utilize unbounded stack allocations, while pull streams need heap allocations for unbounded space in the machine state.

4.6 Agda & Haskell Implementations

In Agda, to avoid computation getting stuck on the generativity axiom, we use the `primTrustMe` built-in [Agda developers 2024] to automatically erase the axiom when the sides of the equation are definitionally equal. Otherwise the implementation closely follows this section.

In Haskell there are some limitations. First, in concatMap, the projection function projS cannot be defined, because Haskell is not dependently typed, and the other field projections are also out of reach. We only have a weaker "positive" recursion principle for existential types. It might be the case that a strongly typed concatMap is possible with only weak existentials, but we attempted this and found that it introduces too much technical complication.

So instead of giving a single generic definition for concatMap for IsSOP types, we define concatMap just for object types,[1] and define $\text{case}_{\text{Pull}}$ separately for each object type, as an overloaded class method. In each of these definitions, we only need to deal with a concrete finite number of machine state types, which is feasible to handle with weak existentials.

Also, the generativity axiom is *false* in Template Haskell, since it is possible to look inside quoted expressions. Instead, we use type coercions that can fail at staging time. If library users do not violate generativity, these coercions disappear and the staging output will not contain unsafe coercions.

5 RELATED WORK

Two-level type theory. Two-level lambda calculi were first developed in the context of abstract interpretation and partial evaluation [Nielson 1984; Nielson and Nielson 1992]. This line of research is characterized by simple types, having the same language features at different stages, and an emphasis on *binding time analysis*, i.e. automatically inferring stage annotations as part of a pipeline for partial evaluation and program optimization.

Later and independently, the same notion of level appeared in homotopy type theory, first in Voevodsky's Homotopy Type System [Voevodsky 2013], and subsequently developed as 2LTT in [Annenkov et al. 2019]. Here, dependent types are essential, and we have different theories at the two stages. The application of 2LTT to staged compilation was developed in [Kovács 2022]. Binding-time analysis has not been developed in this setting; 2LTT-s have been meant to be used as surface languages to directly work in.

We build on both traditions. CFTT itself is a 2LTT and makes heavy use of dependent type theory. However, we are mostly concerned with code generation and optimization and borrow concepts from the partial evaluation literature.

Tracking function arities and closures in types. Downen et al.'s intermediate language $\mathcal{IL}$ has similar motivations as our object language [Downen et al. 2020]. In both systems, function types are distinguished from closure types, enjoy universal η-conversion and have an explicit calling arity. In fact, our object language can be almost viewed as a small simply-typed fragment of $\mathcal{IL}$, with only **letrec** missing from $\mathcal{IL}$.

[1] Recall that we do not distinguish value and computation types in Haskell.

of Gen operations in streams.

```
bind_Gen : IsSOP A ⇒ Gen A → (A → Pull B) → Pull B
bind_Gen g f =
  Pull (Σ A (projS ∘ f)) (do {a ← g; s ← projSeed (f a); return (a, s)}) $ λ case
    (a, s) → projStep (f a) s >>= λ case
      Stop     → return Stop
      Skip s   → return $ Skip (a, s)
      Yield b s → return $ Yield b (a, s)
```

We recover let-insertion and case splitting as follows:

```
gen_Pull : ⇑A → (⇑A → Pull B) → Pull B
gen_Pull a = bind_Gen (gen a)

case_Pull : (Split A, IsSOP (SplitTo {A})) ⇒ ⇑A → (SplitTo {A} → Pull B) → Pull B
case_Pull a = bind_Gen (split a)
```

Let us look at small example. We write forEach for concatMap with flipped arguments.

```
forEach (take ⟨100⟩ (countFrom ⟨0⟩)) $ λ x.
gen_Pull ⟨~x * 2⟩ $ λ y.
case_Pull ⟨~x < 50⟩ $ λ case
  True  → take y (countFrom x)
  False → single y
```

Here, in every forEach iteration, $\text{gen}_{\text{Pull}}\ \langle \sim x * 2 \rangle$ evaluates the given expression and saves the result to the machine state. Then, $\text{case}_{\text{Pull}}$ branches on an object-level Boolean expression. If we use foldr to generate object code from this definition, we get *four* mutually defined functions. We get two state shapes from single y and one from take y (countFrom x); we sum these to get three for the $\text{case}_{\text{Pull}}$, then finally we get an extra shape from forEach which introduces an additional Maybe.

4.5 Discussion

Our stream library has a strong support for programming in a monadic style, even though Pull is not literally a monad. We can bind object values with concatMap, and we can also do let-insertion and case splitting for them. We also get guaranteed well-typing, closure-freedom, and arbitrary mixing of zipping and concatMap.

We highlight the usage of the generativity axiom as well. Previously in staged compilation, intensional analysis (i.e. the ability to analyze object code) has been viewed as a desirable feature that increases the expressive power of a system. To our knowledge, our work is the first one that exploits the *lack* of intensional analysis in metaprogramming. This is a bit similar to parametricity in type theories, where the inability to analyze types has a payoff in the form of "free theorems" [Wadler 1989].

Regarding the practical application of our stream library, we think that it would make sense to support both push and pull fusion in a realistic implementation, and allow users to benefit from the strong points of both. Push streams, which we do not present in this paper, have proper Monad and MonadGen instances and are often more convenient to use in CFTT. They are also better for deep traversals of structures where they can utilize unbounded stack allocations, while pull streams need heap allocations for unbounded space in the machine state.

Sums-of-products. SOP was proposed for generic Haskell programming by De Vries and Löh [de Vries and Löh 2014]. Our own SOP implementation in Haskell is mostly the same as in ibid. The SOP-of-object-code representation appeared as well in typed Template Haskell in [Pickering et al. 2020].

CPS and binding-time improvement. Our Gen monad can be viewed as a variation on the continuation-passing-style that has been often used in staged programming. Flanagan et al.'s ANF translation algorithm uses let-insertion in CPS that works the same way as our gen [Flanagan et al. 1993]. Kiselyov et al. likewise uses essentially the same CPS, in MetaOCaml [Kiselyov et al. 2017]. Jones et al. discuss CPS and binding-time improvement in partial evaluation [Jones et al. 1993]. η-expansion for object-level finite sums is known as "the trick" [Danvy et al. 1996].

Join points. The use join points in GHC's core language is partly motivated by avoiding code duplication in case-of-case transformations [Maurer et al. 2017]. In our monad library, case-of-case is implicitly and eagerly computed during staging, and we similarly use join points to avoid code duplication.

Stream fusion. The staged stream fusion library *strymonas* by Kiselyov et al. is the primary prior art [Kiselyov et al. 2017]. Here, streams are represented as a sum type of push and pull representations, allowing both zipping and concatMap-ing. However, fusion is not guaranteed for all combinations; zipping two concatMap-s reifies one of the streams in a runtime closure. In newer versions of the library, fusion is reported to be complete and guaranteed [Kobayashi and Kiselyov 2024], however, an exposition of this has not yet been published. Another notable difference is that *strymonas* relies on mutable references in the object language, while we do not.

Machine fusion [Robinson and Lippmeier 2017] supports splitting streams to multiple streams while avoiding duplicated computation; this is not possible in *strymonas* or our library. Machine fusion also supports concatMap and zipping. However, its state machine representation is significantly more complicated than ours, and its fusion algorithm is not guaranteed to succeed on arbitrary stream programs.

Coutts developed pull stream fusion in depth in [Coutts 2011]. We borrowed the basic design and the basic stream combinator definitions from there. This account is also close to existing stream implementations in Haskell. Ibid. characterizes fusibility in a non-staged setting, in terms of "good consumers" and "good producers". This scheme does not cover concatMap.

6 CONCLUSIONS AND FUTURE WORK

We believe that a programming language in the style of this paper would be highly useful, especially in high-performance functional programming and domain-specific programming. In particular, the pipeline for using and compiling monads could look like the following.

- Users write definitions in a style similar to Haskell, without quotes, splices, **up**-s and **down**-s, only marking stages in type annotations and in let-definitions, with := and =. Storing monadic actions at runtime requires an explicit boxing operation that yields a closure type.
- Type- and stage-directed elaboration adds the missing operations, and also desugars case splitting using join and the underlying splitting function.
- In the downstream compiler, fast & conservative optimization passes are sufficient, since abstraction overheads are already eliminated by staging. Eliminating dead code and unused arguments would be important, since we have not yet addressed this through staging.
- A systematic way to de-duplicate code would be also needed, probably already during staging.

4.6 Agda & Haskell Implementations

In Agda, to avoid computation getting stuck on the generativity axiom, we use the `primTrustMe` built-in [Agda developers 2024] to automatically erase the axiom when the sides of the equation are definitionally equal. Otherwise the implementation closely follows this section.

In Haskell there are some limitations. First, in concatMap, the projection function projS cannot be defined, because Haskell is not dependently typed, and the other field projections are also out of reach. We only have a weaker "positive" recursion principle for existential types. It might be the case that a strongly typed concatMap is possible with only weak existentials, but we attempted this and found that it introduces too much technical complication.

So instead of giving a single generic definition for concatMap for IsSOP types, we define concatMap just for object types,[1] and define $\text{case}_{\text{Pull}}$ separately for each object type, as an overloaded class method. In each of these definitions, we only need to deal with a concrete finite number of machine state types, which is feasible to handle with weak existentials.

Also, the generativity axiom is *false* in Template Haskell, since it is possible to look inside quoted expressions. Instead, we use type coercions that can fail at unstaging time. If library users do not violate generativity, these coercions disappear and the code output will not contain unsafe coercions.

5 RELATED WORK

Two-level calculi. Two-level lambda calculi were first developed in the context of abstract interpretation and partial evaluation [Nielson 1984; Nielson and Nielson 1992]. This line of research is characterized by simple types, having the same language features at different stages, and an emphasis on *binding time analysis*, i.e. automatically inferring stage annotations as part of a pipeline for partial evaluation and program optimization.

Later and independently, the same notion of level appeared in homotopy type theory, first in Voevodsky's Homotopy Type System [Voevodsky 2013], and subsequently developed as 2LTT in [Annenkov et al. 2019]. Here, dependent types are essential, and we have different theories at the two stages. The application of 2LTT to staged compilation was developed in [Kovács 2022]. Binding-time analysis has not been developed in this setting; 2LTT-s have been meant to be used as surface languages to directly work in.

Higher-order abstract syntax and logical frameworks. Hofmann's work on the semantics of higher-order abstract syntax (HOAS) is an important precursor to our work [Hofmann 1999]. It anticipates many of the later developments in logical frameworks and two-level type theories. In particular, we build on Hofmann's presheaf model for our 2LTT semantics, and his sketch of adequacy for HOAS corresponds to our definition of unstaging and its soundness proof. In general, there is a close correspondence between logical frameworks that represent object languages in HOAS style (e.g. [Harper et al. 1993]) and two-level type theories, and in some cases the two kinds of presentations are inter-derivable [Kovács 2023, Section 3.3]. In fact, our Agda implementation of CFTT also uses a HOAS embedding of the object theory.

Cocon is a logical framework that has a dependently typed meta level ("computation" level) and an object language that is configured by a signature [Pientka et al. 2019]. Cocon, like 2LTTs, can be used as a dependently typed metaprogramming language, but there are several differences. First, Cocon supports and emphasizes intensional analysis (i.e. the ability to do structural induction on object code), which is made possible through a contextual modality [Nanevski et al. 2008]. In contrast, intensional analysis is missing from 2LTTs so far. However, in Cocon, we can only

[1] Recall that we do not distinguish value and computation types in Haskell.

Going a bit further, we believe that a 2LTT-based language could be a good design point for programming in general, buying us plenty of control over code generation for a modest amount of extra complexity.

Also, rebuilding known abstractions in staged programming is valuable because it provides a *semantic explanation* of how abstractions can be compiled, and provides insights that could be reused even in non-staged settings. For instance, in this paper we demonstrated that compiling monadic code can be done by using *the same monads* in the metalanguage, extended with code generation as an effect.

Continuing this line of thought, it could be interesting to also adapt to 2LTT the style of *binding-time analysis* that is well-known in partial evaluation. For example, we might do program optimization in the following way. First, start with monadic code in a non-staged language. Second, try to infer stages, inventing quotes, splices, **up**-s and **down**-s, thereby translating definitions to a 2LTT. Third, perform staging and proceed from there. This would be more fragile than unambiguous stage inference, but we would still benefit from shifting a lot of machinery into staging (which is deterministic and efficient).

We also find it interesting how little impact closure-freedom had on the developments in this paper. This provides some evidence that in staged functional programming, closures can be an opt-in feature instead of the pervasive default.

In this paper we only briefly looked at two applications. Both monad transformers and streams could be further developed, and many other programming abstractions could be revisited in the staged setting.

- We did not discuss the issue of *tail calls* in monadic code. For example, a tail call in Maybe should not case split on the result, while in our current library, making a call with **up** always does so. We did develop an abstraction for tail calls in the Agda and Haskell implementations, but omitted it here partly for lack of space, and partly because we have not yet explored much of the design space.
- In a practical stream library, it would be useful to further try to minimize the size of the machine state. In the Agda and Haskell versions, we additionally track whether streams can Skip, to provide more efficient zipping for non-skipping "synchronous" streams. However, this could be refined in many ways, and we could also try to represent the full transition graph in an observable way, thereby enabling merging or deleting states.
- It seems promising to look for synergies between push streams, pull streams and monad transformers. It seems that pull streams could be generalized from Gen to different monads, in the representation of transitions and initial states. This would allow putting a Pull *on the top* of a monad transformer stack. On the other hand, push streams form a proper monad, but they cannot be used to "transform" other monads, so they can be placed at the *bottom* of a transformer stack.
- We could try to lean more heavily on datatype-generic programming and generalize abstractions to arbitrary object types. For example, push and pull streams could be generalized to inductive and coinductive Church-codings of arbitrary value types.

REFERENCES

Agda developers. 2024. Agda documentation. https://agda.readthedocs.io/en/v2.6.4.2/

Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. 2019. Two-Level Type Theory and Applications. *ArXiv e-prints* (may 2019). http://arxiv.org/abs/1705.03307

Joachim Breitner. 2014. Call Arity. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8843)*, Jurriaan Hage and Jay McCarthy (Eds.). Springer, 34–50. https://doi.org/10.1007/978-3-319-14675-1_3

manipulate object terms at the meta level by specifying their concrete object-level contexts (i.e. possible free variables), while in 2LTTs the object-level contexts are implicit and are only computed during code generation. The latter style is more convenient for staged compilation applications, where explicit context tracking would add a significant amount of bureaucracy.

Nonetheless, it seems to be possible and useful to have a system which simultaneously supports intensional analysis through contextual types and the 2LTT-style implicit contexts.

CPS and monads in staged programming. Writing code generators in continuation-passing style was first proposed by Bondorf [Bondorf 1992]. Flanagan et al.'s ANF translation algorithm uses CPS for let-insertion as well [Flanagan et al. 1993], similarly to our gen function. These sources do no use explicit monadic notation though. Jones et al. discuss CPS and binding-time improvement in partial evaluation in [Jones et al. 1993]. In [Kiselyov et al. 2004], [Swadi et al. 2006] and [Carette and Kiselyov 2011], the composition of state and continuation monads was used, similarly to our StateT *S* Gen, but without polymorphic answer types. The definition of let-insertion here is again similar to ours.

Tracking function arities and closures in types. Downen et al.'s intermediate language $\mathcal{IL}$ has similar motivations as our object language [Downen et al. 2020]. In both systems, function types are distinguished from closure types, enjoy universal η-conversion and have an explicit calling arity. In fact, our object language can be almost viewed as a small simply-typed fragment of $\mathcal{IL}$, with only **letrec** missing from $\mathcal{IL}$.

Sums-of-products. SOP was proposed for generic Haskell programming by De Vries and Löh [de Vries and Löh 2014b]. Our own SOP implementation in Haskell is mostly the same as in ibid. The SOP-of-object-code representation appeared as well in typed Template Haskell in [Pickering et al. 2020]. η-expansion for object-level finite sums is known as "the trick" [Danvy et al. 1996].

Join points. The use join points in GHC's core language is partly motivated by avoiding code duplication in case-of-case transformations [Maurer et al. 2017]. In our monad library, case-of-case is implicitly and eagerly computed during staging, and we similarly use join points to avoid code duplication.

Stream fusion. The staged stream fusion library *strymonas* by Kiselyov et al. is the primary prior art [Kiselyov et al. 2017]. Here, streams are represented as a sum type of push and pull representations, allowing both zipping and concatMap-ing. However, fusion is not guaranteed for all combinations; zipping two concatMap-s reifies one of the streams in a runtime closure. In newer versions of the library, fusion is complete [Kobayashi and Kiselyov 2024], however, an exposition of this has not yet been published. Additionally, *strymonas* heavily relies on mutable references in the object language, and also uses some switching on object-level data to implement control flow. This style of code output can be still reliably optimized by downstream compilers (C or OCaml), but our solution is more conservative in that it does not use any mutation or runtime switching. This "purity" of our streams also makes them easier to generalize, e.g. by using arbitrary MonadGen monads instead of Gen in stream internals.

Machine fusion [Robinson and Lippmeier 2017] supports splitting streams to multiple streams while avoiding duplicated computation; this is not possible in *strymonas* or our library. Machine fusion also supports concatMap and zipping. However, its state machine representation is significantly more complicated than ours, and its fusion algorithm is not guaranteed to succeed on arbitrary stream programs.

Coutts developed pull stream fusion in depth in [Coutts 2011]. We borrowed the basic design and the basic stream combinator definitions from there. This account is also close to existing stream

Duncan Coutts. 2011. *Stream fusion : practical shortcut fusion for coinductive sequence types*. Ph. D. Dissertation. University of Oxford, UK. http://ora.ox.ac.uk/objects/uuid:b4971f57-2b94-4fdf-a5c0-98d6935a44da

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751. https://doi.org/10.1145/236114.236119

Edsko de Vries and Andres Löh. 2014. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 83–94. https://doi.org/10.1145/2633628.2633634

Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds are calling conventions. *Proc. ACM Program. Lang.* 4, ICFP (2020), 104:1–104:29. https://doi.org/10.1145/3408986

Peter Dybjer. 1994. Inductive Families. *Formal Aspects Comput.* 6, 4 (1994), 440–465. https://doi.org/10.1007/BF01211308

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, Janis Voigtländer (Ed.). ACM, 117–130. https://doi.org/10.1145/2364506.2364522

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

GHC developers. 2024a. GHC documentation. https://downloads.haskell.org/ghc/9.8.2/docs/users_guide/

GHC developers. 2024b. GHC.Base source. https://hackage.haskell.org/package/base-4.19.1.0/docs/src/GHC.Base.html#.

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2010. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6647)*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

Jasper Hugunin. 2020. Why Not W?. In *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy (LIPIcs, Vol. 188)*, Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:9. https://doi.org/10.4230/LIPICS.TYPES.2020.8

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. https://doi.org/10.1145/3009837.3009880

Tomoaki Kobayashi and Oleg Kiselyov. 2024. Complete Stream Fusion for Software-Defined Radio. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, 16 January 2024*, Gabriele Keller and Meng Wang (Eds.). ACM, 57–69. https://doi.org/10.1145/3635800.3636962

András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. https://doi.org/10.1145/3547641

András Kovács. 2022. Demo implementation for the paper "Staged Compilation With Two-Level Type Theory". https://doi.org/10.5281/zenodo.6757373

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, Ke Sivaramakrishnan, and Jérôme Vouillon. 2023. The OCaml system release 5.1: Documentation and user's manual. https://v2.ocaml.org/manual/

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer, 228–242. https://doi.org/10.1007/3-540-48959-2_17

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. https://doi.org/10.1145/3062341.3062380

Ieke Moerdijk and Erik Palmgren. 2000. Wellfounded trees in categories. *Ann. Pure Appl. Log.* 104, 1-3 (2000), 189–218. https://doi.org/10.1016/S0168-0072(00)00012-9

mtl developers. 2024. mtl documentation. https://hackage.haskell.org/package/mtl

Flemming Nielson. 1984. *Abstract interpretation using domain theory*. Ph. D. Dissertation. University of Edinburgh, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.350060

Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level functional languages*. Cambridge tracts in theoretical computer science, Vol. 34. Cambridge University Press.

Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, Tom Schrijvers (Ed.). ACM, 122–135. https://doi.org/10.1145/3406088.3409021

implementations in Haskell. We find it interesting that our staged definitions are very close to the non-staged versions there; essentially all of the additional complexity is compartmentalized in our SOP module. Coutts characterized fusibility in a non-staged setting, in terms of "good consumers" and "good producers", but this scheme does not cover concatMap.

6 CONCLUSIONS AND FUTURE WORK

We believe that a programming language in the style of this paper would be highly useful, especially in high-performance functional programming and domain-specific programming. In particular, the pipeline for using and compiling monads could look like the following.

- Users write definitions in a style similar to Haskell, without quotes, splices, **up**-s and **down**-s, only marking stages in type annotations and in let-definitions, with := and =. Storing monadic actions at runtime requires an explicit boxing operation that yields a closure type.
- Type- and stage-directed elaboration adds the missing operations, and also desugars case splitting using join and the underlying splitting function.
- In the downstream compiler, fast & conservative optimization passes are sufficient, since abstraction overheads are already eliminated by staging. Eliminating dead code and unused arguments would be important, since we have not yet addressed this through staging.
- A systematic way to de-duplicate code would be also needed, probably already during staging.

Going a bit further, we believe that a 2LTT-based language could be a good design point for programming in general, buying us plenty of control over code generation for a modest amount of extra complexity.

Also, rebuilding known abstractions in staged programming is valuable because it provides a *semantic explanation* of how abstractions can be compiled, and provides insights that could be reused even in non-staged settings. For instance, in this paper we demonstrated that compiling monadic code can be done by using the same monads in the metalanguage, extended with code generation as an effect.

Continuing this line of thought, it could be interesting to also adapt to 2LTT the style of *binding-time analysis* that is well-known in partial evaluation. For example, we might do program optimization in the following way. First, start with monadic code in a non-staged language. Second, try to infer stages, inventing quotes, splices, **up**-s and **down**-s, thereby translating definitions to a 2LTT. Third, perform staging and proceed from there. This would be more fragile than unambiguous stage inference, but we would still benefit from shifting a lot of machinery into staging (which is deterministic and efficient).

We also find it interesting how little impact closure-freedom had on the developments in this paper. This provides some evidence that in staged functional programming, closures can be an opt-in feature instead of the pervasive default.

In this paper we only briefly looked at two applications. Both monad transformers and streams could be further developed, and many other programming abstractions could be revisited in the staged setting.

- We did not discuss the issue of *tail calls* in monadic code. For example, a tail call in Maybe should not case split on the result, while in our current library, making a call with **up** always does so. We did develop an abstraction for tail calls in the Agda and Haskell implementations, but omitted it here partly for lack of space, and partly because we have not yet explored much of the design space.
- In a practical stream library, it would be useful to further try to minimize the size of the machine state. In the Agda and Haskell versions, we additionally track whether streams can Skip, to provide more efficient zipping for non-skipping "synchronous" streams. However,

Amos Robinson and Ben Lippmeier. 2017. Machine fusion: merging merges, more or less. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, Wim Vanhoof and Brigitte Pientka (Eds.). ACM, 139–150. https://doi.org/10.1145/3131851.3131865

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Vladimir Voevodsky. 2013. A simple type system with two identity types. (2013). Unpublished note.

Philip Wadler. 1989. Theorems for free!. In *Functional Programming Languages and Computer Architecture*. ACM Press, 347–359.

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed template Haskell. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498723

Jeremy Yallop and Oleg Kiselyov. 2019. Generating mutually recursive definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 75–81. https://doi.org/10.1145/3294032.3294078

this could be refined in many ways, and we could also try to represent the full transition graph in an observable way, thereby enabling merging or deleting states.

- It seems promising to look for synergies between push streams, pull streams and monad transformers. It seems that pull streams could be generalized from Gen to different monads, in the representation of transitions and initial states. This would allow putting a Pull *on the top* of a monad transformer stack. On the other hand, push streams form a proper monad, but they cannot be used to "transform" other monads, so they can be placed at the *bottom* of a transformer stack.
- We could try to lean more heavily on datatype-generic programming and generalize abstractions for more object types. For example, push and pull streams could be generalized to inductive and coinductive Church-codings of arbitrary value types.