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), 21 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

Without optimizations we roughly get:

$$\begin{aligned} f = \lambda\, x.\ (\gg\!=)\ \mathsf{MonadReaderDict}\ \mathsf{ask}\ (\lambda\, b.\ \mathbf{if}\ b \\ \mathbf{then\ return}\ \ \mathsf{MonadReaderDict}\ (x + 10) \\ \mathbf{else\ \ return}\ \ \mathsf{MonadReaderDict}\ (x + 20)) \end{aligned}$$

Here, MonadReaderDict is a runtime dictionary, containing the methods of the Monad instance for Reader, and $(\gg\!=)$ MonadReaderDict is a field projection. Here, a runtime closure will be created for the $\lambda\, b.\ \ldots$ function, and $(\gg\!=)$, 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:

$$\mathsf{mapM} :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$$

This is a third-order and second-rank polymorphic function in disguise, because its monad dictionary argument contains the polymorphic second-order $(\gg\!=)$ 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 good 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, but without any strong guarantee, 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 $(.)\ f\ g = \lambda\, x \to f\ (g\ x)$ in the base libraries, instead of as $(.)\ f\ g\ x = f\ (g\ x)$, to get better inlining behavior [?]. It remains a common practice in high-performance Haskell programming to manually 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.

The idea of **two-level type theory** (2LTT) is to use a highly expressive dependent type theory for compile-time computation, but generate code in possibly different, simpler object language. 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.

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 standard Martin-Löf 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 focus on 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.
- Functors and first-class modules in OCaml 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 **??** we present the two-level type theory CFTT, where the object language is first-order simply-typed and the metalanguage is dependently typed. The object language admits an operational semantics without runtime closures, and can be compiled in a way such that all function calls are statically known. We provide a supplementary Agda formalization of the operational semantics.
- In Section **??** 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 the generality of monad transformers.
- In Section **??** we build a pull-based stream fusion library. Here, the motivation is to demonstrate essential usage of dependent types to concisely solve a difficult problem, namely guaranteed fusion for unrestricted 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 a Template Haskell, 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 more faithful Agda embedding of CFTT and our libraries. 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 focus on informal explanations and examples here. 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, products and indexed inductive types [**?**].

Formally, MetaTy is additionally indexed by universes levels, and we have $\mathsf{MetaTy}_i : \mathsf{MetaTy}_{i+1}$. However, this adds a bit of noise, and it is not very relevant to the current paper, so we shall omit levels. Note that universe levels are orthogonal to staging; they are instead used to increase the strength of the system while maintaining logical consistency.

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:

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

$$\begin{aligned}\textbf{data}\ \mathsf{Bool}_\mathsf{M} &: \mathsf{MetaTy}\ \textbf{where}\\ \mathsf{True}_\mathsf{M} &: \mathsf{Bool}_\mathsf{M}\\ \mathsf{False}_\mathsf{M} &: \mathsf{Bool}_\mathsf{M}\end{aligned}$$

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 **newtype** Wrap A = Wrap {unWrap : A}, and also a similar notation for (dependent) record types, for instance as in

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

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. Similarly as in the case of MetaTy, all construction and elimination rules of the object language stay within Ty. However, 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
```

Every recursive definition must be 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 as well, 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 is *saturated*. This means that every function call is of the form $\mathsf{f}\ \mathsf{t}_1\ \mathsf{t}_2 \ldots \mathsf{t}_n$, 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 we may 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), and there are similarities, 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 us. In future work we could make a closure look at the connections to CBPV.

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 $A : \mathsf{Ty}$, we have $\Uparrow A : \mathsf{MetaTy}$, pronounced as "lift A". This is the type of metaprograms that produce A-typed object programs.
- For $A : \mathsf{Ty}$ and $t : A$, we have $\langle t \rangle : \Uparrow A$, pronounced "quote t". This is the metaprogram which immediately returns t.
- For $t : \Uparrow A$, we have $\sim t : A$, pronounced "splice t". This inserts the result of a metaprogram into an object term. *Notation:* splicing binds stronger than function application, so $f \sim x$ is parsed as $f\ (\sim x)$.
- We have $\langle \sim t \rangle \equiv t$ and $\sim \langle t \rangle \equiv t$ 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. This staging is formalized in detail in [**?**]. In Section **??** we describe the modifications to ibid. that are used in this paper.

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:

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

Here, id is used at type $\Uparrow\mathsf{Nat}$. During staging, the expression in the splice is evaluated, so we get $\sim\langle 10 + 10 \rangle$, which is definitionally the same as $10 + 10$, which is our staging output here. Boolean short-circuiting is another basic use-case:

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

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 with generally 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 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.

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 ⇑A → ⇑B is usually preferable to ⇑(A → 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:

```
up : ⇑(A → B) → ⇑A → ⇑B        down : (⇑A → ⇑B) → ⇑(A → B)
up f a = ⟨~f ~a⟩               down f = ⟨λ a. ~(f ⟨a⟩)⟩
```

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. 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:

```
up : ⇑(A, B) → (⇑A, ⇑B)        down : (⇑A, ⇑B) → ⇑(A, B)
up x = (⟨fst ~x⟩, ⟨snd ~x⟩)     down (x, y) = ⟨(~x, ~y)⟩
```

Here we overload Haskell-style product type notation, both for types and the pair constructor, at both levels (products are definable as a value type at the object level). 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. Fortunately, there is a principled solution.

3.2 The Code Generation Monad

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

$$\textbf{newtype}\ \mathsf{Gen}\ \mathsf{A} = \mathsf{Gen}\ \{\mathsf{unGen} : \{\mathsf{R} : \mathsf{Ty}\} \to (\mathsf{A} \to \Uparrow \mathsf{R}) \to \Uparrow \mathsf{R}\}$$

This is a monad in MetaTy in a standard sense:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monad}\ \mathsf{Gen}\ \textbf{where}\\ &\quad \textbf{return}\ \mathsf{a} = \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\ \mathsf{k}\ \mathsf{a}\\ &\quad \mathsf{ga} \gg\!= \mathsf{f}\ \ = \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\ \mathsf{unGen}\ \mathsf{ga}\ (\lambda\, \mathsf{a}.\ (\mathsf{f}\ \mathsf{a}\ \mathsf{k}))\end{aligned}$$

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 Haskell implementation of the paper, and also in the Agda implementation, 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:

$$\begin{aligned}&\mathsf{runGen} : \mathsf{Gen}\ (\Uparrow \mathsf{A}) \to \Uparrow \mathsf{A}\\ &\mathsf{runGen}\ \mathsf{ma} = \mathsf{unGen}\ \mathsf{ma}\ \mathsf{id}\end{aligned}$$

We can let-bind object expressions in Gen:

$$\begin{aligned}&\mathsf{gen} : \Uparrow \mathsf{A} \to \mathsf{Gen}\ (\Uparrow \mathsf{A})\\ &\mathsf{gen}\ \mathsf{a} = \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\ \langle \textbf{let}\ \mathsf{x} : \mathsf{A} := \sim \mathsf{a}; \sim (\mathsf{k}\ \langle \mathsf{x} \rangle) \rangle\end{aligned}$$

And also recursive definitions of computations:

$$\begin{aligned}&\mathsf{genRec} : \{\mathsf{A} : \mathsf{CompTy}\} \to (\Uparrow \mathsf{A} \to \Uparrow \mathsf{A}) \to \mathsf{Gen}\ (\Uparrow \mathsf{A})\\ &\mathsf{genRec}\ \mathsf{f} = \mathsf{Gen}\ \ \$\ \lambda\, \mathsf{k}.\ \langle \textbf{letrec}\ \mathsf{x} : \mathsf{A} := \sim (\mathsf{f}\ \langle \mathsf{x} \rangle); \sim (\mathsf{k}\ \langle \mathsf{x} \rangle) \rangle\end{aligned}$$

Now, using do-notation, we may write $\textbf{do}\ \{\mathsf{x} \leftarrow \mathsf{gen}\ \langle 10 + 10 \rangle; \mathsf{y} \leftarrow \mathsf{gen}\ \langle 20 + 20 \rangle\}; \textbf{return}\ \langle \mathsf{x} + \mathsf{y} \rangle\}$, for a Gen ($\Uparrow$Nat) action. Running this with runGen yields $\langle \textbf{let}\ \mathsf{x} := 10 + 10; \textbf{let}\ \mathsf{y} := 20 + 20; \mathsf{x} + \mathsf{y} \rangle$. We can also define a "safer" binding-time improvement for products, using let-insertion:

$$\begin{aligned}&\mathsf{up} : \Uparrow(\mathsf{A},\ \mathsf{B}) \to \mathsf{Gen}\ (\Uparrow \mathsf{A},\ \Uparrow \mathsf{B})\\ &\mathsf{up}\ \mathsf{x} = \textbf{do}\ \mathsf{x} \leftarrow \mathsf{gen}\ \mathsf{x}\\ &\qquad\qquad \textbf{return}\ (\langle \mathsf{fst}\ \sim \mathsf{x} \rangle,\ \langle \mathsf{snd}\ \sim \mathsf{x} \rangle)\}\end{aligned}$$

$$\begin{aligned}&\mathsf{down} : \mathsf{Gen}\ (\Uparrow \mathsf{A},\ \Uparrow \mathsf{B}) \to \Uparrow(\mathsf{A},\ \mathsf{B})\\ &\mathsf{down}\ x = \textbf{do}\ (\mathsf{a},\ \mathsf{b}) \leftarrow x\\ &\qquad\qquad\qquad \textbf{return}\ \langle (\sim \mathsf{a},\ \sim \mathsf{b}) \rangle\end{aligned}$$

Working in Gen is convenient, since we can freely generate object code and also have access to the full metalanguage. Also, the whole point of staging is that *eventually* all metaprograms will be used for the purpose of code generation, so ultimately we always want to runGen our actions. So why not just always work in Gen? The implicitness 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 prefer to not write most of our program in IO.

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:

$$(\gg\!=) : \Uparrow(\mathsf{Maybe}\ \mathsf{A}) \to (\Uparrow \mathsf{A} \to \Uparrow(\mathsf{Maybe}\ \mathsf{B})) \to \Uparrow(\mathsf{Maybe}\ \mathsf{B})$$

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:

$$\begin{aligned}&\textbf{class}\ \mathsf{MonadGen}\ \mathsf{M} \Rightarrow \mathsf{Improve}\ (\mathsf{F} : \mathsf{ValTy} \to \mathsf{Ty})\ (\mathsf{M} : \mathsf{MetaTy} \to \mathsf{MetaTy})\ \textbf{where}\\ &\quad \mathsf{up}\ \ \ : \{\mathsf{A} : \mathsf{ValTy}\} \to \Uparrow(\mathsf{F}\ \mathsf{A}) \to \mathsf{M}\ (\Uparrow \mathsf{A})\\ &\quad \mathsf{down} : \{\mathsf{A} : \mathsf{ValTy}\} \to \mathsf{M}\ (\Uparrow \mathsf{A}) \to \Uparrow(\mathsf{F}\ \mathsf{A})\end{aligned}$$

Assume that $\mathsf{Maybe}_\mathsf{M}$ is the standard meta-level monad, and $\mathsf{MaybeT}_\mathsf{M}$ is the standard monad transformer:

$$\textbf{newtype}\ \mathsf{MaybeT}_\mathsf{M}\ \mathsf{M}\ \mathsf{A} = \mathsf{MaybeT}_\mathsf{M}\ \{\mathsf{runMaybeT}_\mathsf{M} : \mathsf{M}\ (\mathsf{Maybe}_\mathsf{M}\ \mathsf{A})\}$$

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

$$\begin{aligned}&\textbf{instance}\ \mathsf{Improve}\ \mathsf{Maybe}\ (\mathsf{MaybeT}_\mathsf{M}\ \mathsf{Gen})\ \textbf{where}\\ &\quad \mathsf{up}\ \mathsf{x} = \mathsf{MaybeT}_\mathsf{M}\ \$\ \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\\ &\qquad \langle \textbf{case}\ {\sim}\mathsf{x}\ \textbf{of}\ \mathsf{Nothing} \to {\sim}(\mathsf{k}\ \mathsf{Nothing}_\mathsf{M}); \mathsf{Just}\ \mathsf{a} \to {\sim}(\mathsf{k}\ (\mathsf{Just}_\mathsf{M} \langle \mathsf{a} \rangle))\rangle\\ &\quad \mathsf{down}\ (\mathsf{MaybeT}_\mathsf{M}\ (\mathsf{Gen}\ \mathsf{ma})) =\\ &\qquad \mathsf{ma}\ (\lambda\, \mathsf{x}.\ \textbf{case}\ \mathsf{x}\ \textbf{of}\ \mathsf{Nothing}_\mathsf{M} \to \langle \mathsf{Nothing} \rangle; \mathsf{Just}_\mathsf{M}\ \mathsf{a} \to \langle \mathsf{Just}\ {\sim}\mathsf{a} \rangle)\end{aligned}$$

With this, we get the Monad instance for free from $\mathsf{MaybeT}_\mathsf{M}$ and Gen. A small example:

$$\textbf{let}\ \mathsf{n} : \mathsf{Maybe}\ \mathsf{Nat} := {\sim}(\mathsf{down}\ \$\ \textbf{do}\ \{\mathsf{x} \leftarrow \textbf{return}\ \langle 10 \rangle; \mathsf{y} \leftarrow \textbf{return}\ \langle 20 \rangle; \textbf{return}\ \langle \mathsf{x} + \mathsf{y} \rangle\}); \ldots$$

Since $\mathsf{MaybeT}_\mathsf{M}$ is meta-level, its monadic binding fully computes at staging time. Thus, the above code is staged to

$$\textbf{let}\ \mathsf{n} : \mathsf{Maybe}\ \mathsf{Nat} := \mathsf{Just}\ (10 + 20);\ \ldots$$

Assume also a $\mathsf{lift} : \mathsf{Monad}\ \mathsf{M} \Rightarrow \mathsf{M}\ \mathsf{A} \to \mathsf{MaybeT}_\mathsf{M}\ \mathsf{M}\ \mathsf{A}$ operation which comes from $\mathsf{MaybeT}_\mathsf{M}$ being a monad transformer. We can do let-insertion in $\mathsf{MaybeT}_\mathsf{M}$ Gen by simply lifting:

$$\begin{aligned}&\mathsf{gen}' : \Uparrow \mathsf{A} \to \mathsf{MaybeT}_\mathsf{M}\ \mathsf{Gen}\ (\Uparrow \mathsf{A})\\ &\mathsf{gen}'\ \mathsf{a} = \mathsf{lift}\ (\mathsf{gen}\ \mathsf{a})\end{aligned}$$

However, it is more convenient to proceed in the style of Haskell's monad transformer library [**?**], and have a class for monads that can do code generation:

$$\begin{aligned}&\textbf{class}\ \mathsf{Monad}\ \mathsf{M} \Rightarrow \mathsf{MonadGen}\ \mathsf{M}\ \textbf{where}\\ &\quad \mathsf{liftGen} : \mathsf{Gen}\ \mathsf{A} \to \mathsf{M}\ \mathsf{A}\\ &\textbf{instance}\ \mathsf{MonadGen}\ \mathsf{Gen}\ \textbf{where}\ \mathsf{liftGen} = \mathsf{id}\\ &\textbf{instance}\ \mathsf{MonadGen}\ \mathsf{M} \Rightarrow \mathsf{MonadGen}\ (\mathsf{MaybeT}_\mathsf{M}\ \mathsf{M})\ \textbf{where}\ \mathsf{liftGen} = \mathsf{lift} \circ \mathsf{liftGen}\end{aligned}$$

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:

$$\begin{aligned}\mathsf{gen} &: \mathsf{MonadGen}\ \mathsf{M} \Rightarrow \Uparrow \mathsf{A} \to \mathsf{M}\ (\Uparrow \mathsf{A})\\ \mathsf{genRec} &: \mathsf{MonadGen}\ \mathsf{M} \Rightarrow (\Uparrow \mathsf{A} \to \Uparrow \mathsf{A}) \to \mathsf{M}\ (\Uparrow \mathsf{A})\end{aligned}$$

3.3.1 Case splitting in monads. We often want to case-split on object-level data inside a monadic action, like in the following:

$$\begin{aligned}&\mathsf{f} : \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\ &\mathsf{f\ x} := {\sim}(\mathsf{down}\ \$\ \mathbf{case}\ \mathsf{x} == 10\ \mathbf{of}\\ &\quad \mathsf{True} \to \mathbf{return}\ \langle \mathsf{x} + 5 \rangle;\\ &\quad \mathsf{False} \to \mathsf{Nothing_M})\end{aligned}$$

This is ill-typed as written, since we cannot compute meta-level actions from an object-level case split. Fortunately, with a little bit more work, case splitting on object values is actually possible in any MonadGen, on any value type.

We demonstrate this for lists first. 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:

$$\mathbf{data}\ \mathsf{SplitList\ A} = \mathsf{Nil'} \mid \mathsf{Cons'}\ (\Uparrow \mathsf{A})\ (\Uparrow (\mathsf{List\ A}))$$

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

$$\begin{aligned}&\mathsf{split} : \Uparrow (\mathsf{List\ A}) \to \mathsf{Gen}\ (\mathsf{SplitList\ A})\\ &\mathsf{split\ as} = \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\ \langle \mathbf{case}\ {\sim}\mathsf{as}\ \mathbf{of}\ \mathsf{Nil} \to {\sim}(\mathsf{k\ Nil'}); \mathsf{Cons\ a\ as} \to {\sim}(\mathsf{k}\ (\mathsf{Cons'}\ \langle a \rangle\ \langle \mathsf{as} \rangle))\rangle\end{aligned}$$

Now, in any MonadGen, we may write

$$\mathbf{do}\ \{\mathsf{sp} \leftarrow \mathsf{liftGen}\ (\mathsf{split\ as}); (\mathbf{case}\ \mathsf{sp}\ \mathbf{of}\ \mathsf{Nil'} \to \ldots; \mathsf{Cons'\ a\ as} \to \ldots)\}$$

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:

$$\begin{aligned}&\mathbf{class}\ \mathsf{Split}\ (\mathsf{A} : \mathsf{ValTy})\ \mathbf{where}\\ &\quad \mathsf{SplitTo} : \mathsf{MetaTy}\\ &\quad \mathsf{split} \quad : \Uparrow \mathsf{A} \to \mathsf{Gen\ SplitTo}\end{aligned}$$

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 we will be able to write **case** x **of** ... in a **do** -block, whenever we work in a MonadGen and x has a value type.

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:

$$\begin{aligned}&\mathbf{newtype}\ \mathsf{Identity\ A} := \mathsf{Identity}\ \{\mathsf{runIdentity} : \mathsf{A}\}\\ &\mathbf{instance}\ \mathsf{Improve\ Identity\ Gen}\ \mathbf{where}\\ &\quad \mathsf{up\ x} = \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\ \mathsf{k}\ \langle \mathsf{runIdentity}\ {\sim}\mathsf{x} \rangle\\ &\quad \mathsf{down\ x} = \mathsf{unGen\ x}\ \$\ \lambda\, \mathsf{a}.\ \langle \mathsf{Identity}\ {\sim}\mathsf{a} \rangle\end{aligned}$$

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

$$\textbf{newtype}\ \mathsf{MaybeT}\ (\mathsf{M} : \mathsf{ValTy} \to \mathsf{Ty})\ (\mathsf{A} : \mathsf{ValTy}) := \mathsf{MaybeT}\ \{\mathsf{runMaybeT} : \mathsf{M}\ (\mathsf{Maybe}\ \mathsf{A})\}$$

With this, improvement can be generally defined for MaybeT:

$$\begin{aligned}
&\textbf{instance}\ \mathsf{Improve}\ \mathsf{F}\ \mathsf{M} \Rightarrow \mathsf{Improve}\ (\mathsf{MaybeT}\ \mathsf{F})\ (\mathsf{MaybeT}_\mathsf{M}\ \mathsf{M})\ \textbf{where}\\
&\quad \mathsf{up}\ \mathsf{x} = \mathsf{MaybeT}_\mathsf{M}\ \$\ \textbf{do}\\
&\qquad \mathsf{ma} \leftarrow \mathsf{up}\ \langle \mathsf{runMaybeT}\ {\sim}\mathsf{x} \rangle\\
&\qquad \textbf{case}\ \mathsf{ma}\ \textbf{of}\ \mathsf{Nothing} \to \textbf{return}\ \mathsf{Nothing}_\mathsf{M}\\
&\qquad\qquad\quad \mathsf{Just}\ \mathsf{a} \to \textbf{return}\ (\mathsf{Just}_\mathsf{M}\ \mathsf{a})\\
&\quad \mathsf{down}\ (\mathsf{MaybeT}_\mathsf{M}\ \mathsf{x}) = \langle \mathsf{MaybeT}\ {\sim}(\mathsf{down}\ \$\ \mathsf{x} \ggg\!= \lambda\ \textbf{case}\\
&\qquad \mathsf{Nothing}_\mathsf{M} \to \textbf{return}\ \langle \mathsf{Nothing} \rangle\\
&\qquad \mathsf{Just}_\mathsf{M}\ \mathsf{a} \to \textbf{return}\ \langle \mathsf{Just}\ {\sim}\mathsf{a} \rangle) \rangle
\end{aligned}$$

In the **case** in up, we already 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 Haskell's monad transformer library mtl. 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 present only StateT and ReaderT. Starting with StateT, we assume $\mathsf{StateT}_\mathsf{M}$ as the standard meta-level definition. The object-level StateT has type $(\mathsf{S} : \mathsf{ValTy})(\mathsf{F} : \mathsf{ValTy} \to \mathsf{Ty})(\mathsf{A} : \mathsf{ValTy}) \to \mathsf{ValTy}$; the state parameter S has to be a value type, since it is an input to an object-level function.

$$\begin{aligned}
&\textbf{instance}\ \mathsf{Improve}\ \mathsf{F}\ \mathsf{M} \Rightarrow \mathsf{Improve}\ (\mathsf{StateT}\ \mathsf{S}\ \mathsf{F})\ (\mathsf{StateT}_\mathsf{M}\ (⇑\mathsf{S})\ \mathsf{M})\ \textbf{where}\\
&\quad \mathsf{up}\ \mathsf{x} = \mathsf{StateT}_\mathsf{M}\ \$\ \lambda\ \mathsf{s}.\ \textbf{do}\\
&\qquad \mathsf{as} \leftarrow \mathsf{up}\ \langle \mathsf{runStateT}\ {\sim}\mathsf{x}\ {\sim}\mathsf{s} \rangle\\
&\qquad \textbf{case}\ \mathsf{as}\ \textbf{of}\ (\mathsf{a},\ \mathsf{s}) \to \textbf{return}\ (\mathsf{a},\ \mathsf{s})\\
&\quad \mathsf{down}\ \mathsf{x} = \langle \mathsf{StateT}\ (\lambda\ \mathsf{s}.\ {\sim}(\mathsf{down}\ \$\ \textbf{do}\\
&\qquad (\mathsf{a},\ \mathsf{s}) \leftarrow \mathsf{runStateT}_\mathsf{M}\ \mathsf{x}\ \langle \mathsf{s} \rangle\\
&\qquad \textbf{return}\ \langle ({\sim}\mathsf{a},\ {\sim}\mathsf{s}) \rangle))) \rangle
\end{aligned}$$

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.

$$\begin{aligned}
&\textbf{instance}\ \mathsf{Improve}\ \mathsf{F}\ \mathsf{M} \Rightarrow \mathsf{Improve}\ (\mathsf{ReaderT}\ \mathsf{R}\ \mathsf{F})\ (\mathsf{ReaderT}_\mathsf{M}\ (⇑\mathsf{R})\ \mathsf{M})\ \textbf{where}\\
&\quad \mathsf{up}\ \mathsf{x} = \mathsf{ReaderT}_\mathsf{M}\ \$\ \lambda\ \mathsf{r}.\ \mathsf{up}\ \langle \mathsf{runReaderT}\ {\sim}\mathsf{x}\ {\sim}\mathsf{r} \rangle\\
&\quad \mathsf{down}\ \mathsf{x} = \langle \mathsf{ReaderT}\ (\lambda\ \mathsf{r}.\ {\sim}(\mathsf{down}\ (\mathsf{runReaderT}_\mathsf{M}\ \mathsf{x}\ \langle \mathsf{r} \rangle))) \rangle
\end{aligned}$$

3.4.1 State *and* Reader *operations.* If we try to use the modify function that we already have for $\mathsf{State}_\mathsf{M}$, a curious thing happens. The meaning of modify $(\lambda\ \mathsf{x}.\ \langle {\sim}\mathsf{x} + {\sim}\mathsf{x} \rangle)$ is to replace the current state x, as an object expression, with the expression $\langle {\sim}\mathsf{x} + {\sim}\mathsf{x} \rangle$, 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,

$$\mathsf{down}\ \$\ \textbf{do}\ \{\mathsf{modify}\ (\lambda\ \mathsf{x}.\ \langle {\sim}\mathsf{x} + {\sim}\mathsf{x} \rangle); \mathsf{modify}\ (\lambda\ \mathsf{x}.\ \langle {\sim}\mathsf{x} + {\sim}\mathsf{x} \rangle); \textbf{return}\ \langle () \rangle\}$$

is staged to

$$\langle \lambda\ \mathsf{x}.\ ((),\ (\mathsf{x} + \mathsf{x}) + (\mathsf{x} + \mathsf{x})) \rangle$$

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 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 as an object value, consider:

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

The modify′(λ x.⟨~x + 10⟩) action gets inlined to 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 **case** splits sequenced after each other, that yields exponential code size. Right now, we can fix this by let-binding the problematic action:

```
down $ do
  x ← gen $ down $ case b of
    True  → put′⟨10⟩
    False → put′⟨20⟩
  up x
  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, since 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 ()
```

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 $\{R : \mathsf{ValTy}\} \to (A \to \Uparrow R) \to \Uparrow R$. Here, if A is a finite sum, we can rearrange $A \to \Uparrow R$ 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 **??**, 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).

$$\mathsf{U_{SOP}} : \mathsf{MetaTy} \qquad \mathsf{El_{SOP}} : \mathsf{U_{SOP}} \to \mathsf{MetaTy}$$
$$\mathsf{U_{SOP}} = \mathsf{List}\,(\mathsf{List}\,\mathsf{ValTy}) \qquad \mathsf{El_{SOP}}\,A = \ldots$$

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. SOP is closed under value types, finite product types and finite sum types. For instance, we have $\mathsf{Either_{SOP}} : \mathsf{SOP} \to \mathsf{SOP} \to \mathsf{SOP}$ together with $\mathsf{Left_{SOP}} : \mathsf{El_{SOP}}\,A \to \mathsf{El_{SOP}}\,(\mathsf{Either_{SOP}}\,A\,B)$, $\mathsf{Right_{SOP}} : \mathsf{El_{SOP}}\,B \to \mathsf{El_{SOP}}\,(\mathsf{Either_{SOP}}\,A\,B)$ 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, $A \simeq \mathsf{Rep}$ 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 $A \to \mathsf{Rep}$, and write rep^{-1} for its inverse. Now we define an isomorphic representation of $\mathsf{El_{SOP}}\,A \to \Uparrow R$ as a product of object-level functions:

$$\mathsf{Fun_{SOP\Uparrow}} : \mathsf{U_{SOP}} \to \mathsf{Ty} \to \mathsf{MetaTy}$$
$$\mathsf{Fun_{SOP\Uparrow}}\,\mathsf{Nil}\qquad R = ()$$
$$\mathsf{Fun_{SOP\Uparrow}}\,(\mathsf{Cons}\,A\,B)\,R = (\Uparrow(\mathsf{foldr}\,(\to)\,R\,A),\ \mathsf{Fun_{SOP\Uparrow}}\,B\,R)$$

$$\mathsf{tabulate} : (\mathsf{El_{SOP}}\,A \to \Uparrow R) \to \mathsf{Fun_{SOP\Uparrow}}\,A\,R$$
$$\mathsf{index} \quad : \mathsf{Fun_{SOP\Uparrow}}\,A\,R \to (\mathsf{El_{SOP}}\,A \to \Uparrow R)$$

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

$$\mathsf{genFun_{SOP\Uparrow}} : \{A : \mathsf{U_{SOP}}\} \to \mathsf{Fun_{SOP\Uparrow}}\,A\,R \to \mathsf{Fun_{SOP\Uparrow}}\,A\,R$$
$$\mathsf{genFun_{SOP\Uparrow}}\,\{\mathsf{Nil}\}\qquad ()\qquad = \mathbf{return}\ ()$$
$$\mathsf{genFun_{SOP\Uparrow}}\,\{\mathsf{Cons}\ \_\ A\}\,(f,\ fs) = (,) <\$> \mathsf{gen}\ f <*> \mathsf{genFun_{SOP\Uparrow}}\,\{A\}\ fs$$

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

$$\textbf{class}\ \mathsf{Monad}\ \mathsf{M} \Rightarrow \mathsf{MonadJoin}\ \mathsf{M}\ \textbf{where}\ \mathsf{join} : \mathsf{IsSOP}\ \mathsf{A} \Rightarrow \mathsf{M}\ \mathsf{A} \to \mathsf{M}\ \mathsf{A}$$

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

$$\begin{aligned}
&\textbf{instance}\ \mathsf{MonadJoin}\ \mathsf{Gen}\ \textbf{where}\\
&\quad \mathsf{join}\ \mathsf{ma} = \mathsf{Gen}\ \$\ \lambda\, \mathsf{k}.\ \mathsf{runGen}\ \$\ \textbf{do}\\
&\qquad \mathsf{joinPoints} \leftarrow \mathsf{genFun}_{\mathsf{SOP}\Uparrow}\ (\mathsf{tabulate}\ (\mathsf{k} \circ \mathsf{rep}^{-1}))\\
&\qquad \mathsf{a} \leftarrow \mathsf{ma}\\
&\qquad \textbf{return}\ \$\ \mathsf{index}\ \mathsf{joinPoints}\ (\mathsf{rep}\ \mathsf{a})
\end{aligned}$$

Here we first convert $\mathsf{k} : \mathsf{A} \to \Uparrow \mathsf{R}$ 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 $\mathsf{StateT}_\mathsf{M}$, we need the extra IsSOP S constraint because S is returned as a result value.

$$\begin{aligned}
&\textbf{instance}\ (\mathsf{MonadJoin}\ \mathsf{M}) \Rightarrow \mathsf{MonadJoin}\ (\mathsf{MaybeT}_\mathsf{M}\ \mathsf{M})\ \textbf{where}\\
&\quad \mathsf{join}\ (\mathsf{MaybeT}_\mathsf{M}\ \mathsf{ma}) = \mathsf{MaybeT}_\mathsf{M}\ (\mathsf{join}\ \mathsf{ma})\\
&\textbf{instance}\ (\mathsf{MonadJoin}\ \mathsf{M}) \Rightarrow \mathsf{MonadJoin}\ (\mathsf{ReaderT}_\mathsf{M}\ \mathsf{R}\ \mathsf{M})\ \textbf{where}\\
&\quad \mathsf{join}\ (\mathsf{ReaderT}_\mathsf{M}\ \mathsf{ma}) = \mathsf{ReaderT}_\mathsf{M}\ (\mathsf{join} \circ \mathsf{ma})\\
&\textbf{instance}\ (\mathsf{MonadJoin}\ \mathsf{M},\ \mathsf{IsSOP}\ \mathsf{S}) \Rightarrow \mathsf{MonadJoin}\ (\mathsf{StateT}_\mathsf{M}\ \mathsf{S}\ \mathsf{M})\ \textbf{where}\\
&\quad \mathsf{join}\ (\mathsf{StateT}_\mathsf{M}\ \mathsf{ma}) = \mathsf{StateT}_\mathsf{M}\ (\mathsf{join} \circ \mathsf{ma})
\end{aligned}$$

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.

bigger example, non-tail-recursive list function with some effects, input, output, mentioned tail rec

3.6 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 switching 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 code noise can be mitigated very effectively in a native CFTT implementation. Kovács [**?**] demonstrated in a prototype elaborator that almost all quotes and splices are unambiguously inferable in 2LTT programs, 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 stage inference and binding-time improvement 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.

3.7 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. We do get high quality code though, typically.
- 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* [**?**] 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 [**?**]. 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 IsSumVS 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).

Let us see some operations on streams now. First, Pull is a "zippy" applicative functor, where pure infinitely repeats a value and (<*>) applies a stream of functions to a stream of values, pointwise.

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

(<*>) : Pull (A → B) → Pull A → Pull B
(<*>) (Pull S seed step) (Pull S′ seed′ step′) =
  Pull (S, S′) ((,) <$> seed <*> seed′) $ λ (s, s′). step s >>= λ case
    Stop     → return Stop
    Skip s   → return $ Skip (s, s′)
    Yield f s → step′ s′ >>= λ case
      Stop     → return Stop
      Skip s′  → return $ Skip (s, s′)
      Yield a s′ → return $ Yield (f a) (s, s′)
```

In pure, the state is the unit type, while in (<*>) we take the product of the stream states and do synchronous transitions. In both definitions, the IsSOP S constraint is implicitly dispatched by instance resolution; this works in the Agda and Haskell versions too. We can derive the usual zip and zipWith operations from (<*>). 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; note though that 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 S state 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 → Gen 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

[?]. 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 () : CompTy and (,) : CompTy → CompTy → 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 → B, A → 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 gets compiled to a sequence of function definitions. Now, we can 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)));
  ~(runGen $ do {s ← seed; return $ index ⟨fs⟩ (rep s)})⟩
```

This foldr is quite flexible, becuse we can eliminate into any object type, including function 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 required in the compiler to eliminate the intermediate closures [?].

4.3 concatMap for Streams

We saw that Pull is Applicative, but what about having a list-like Monad instance as well, with singleton streams for **return** and concatMap for binding? This is not possible. Aiming at

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

the $\mathsf{A} \to \mathsf{Pull\ B}$ function can contain an infinite number of different machine state types, which cannot be represented in a finite amount of object code.[1] Here by "infinite" we mean the notion that is internally available in the meta type theory. For instance, we can define a $\mathsf{Nat}_\mathsf{M} \to \mathsf{Pull\ B}$ function which for each $\mathsf{n} : \mathsf{Nat}_\mathsf{M}$ produces a concatenation of n streams. Hence, we shall have the following function instead:

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

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

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

Above, $\{\mathsf{a} : \mathsf{A}\} \to \mathsf{IsSOP\ (B\ a)}$ is a universally quantified instance constraint; this is available in Agda and, less generally, in Haskell [**?**]. 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 $\mathsf{a} : \mathsf{A}$ value and we run the $(\mathsf{f\ a})$ stream until it stops. In the inner loop, the machine state type depends on the $\mathsf{a} : \mathsf{A}$ 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.

[1] Although in two-level type theories that are not used for staging, the representability of countably infinite products of object types is sometimes assumed as an axiom; this axiom is called "cofibrancy of $\mathbb{N}$" [**?**].

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

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

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

Axiom (generativity). *Every* $f : \mathsf{El_P\ A} \to \mathsf{U_{SOP}}$ *is constant.*

We remark that there is no risk of staging getting *stuck* on this axiom, because propositional equality proofs get *erased* in the staging semantics, as we noted in Section **??**.

We also remark that although generativity is inconsistent with inspecting the structure of object terms, it is consistent with inspecting the structure of object types.

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

Then, $\Sigma_{\mathsf{SOP}}\ \mathsf{A\ B}$ is defined as the concatenation of $\mathsf{A}_i \times \mathsf{B}\ (\mathsf{inject}_i\ \mathsf{loop}_{\mathsf{A}_i})$ for each $\mathsf{A}_i \in \mathsf{A}$, where $(\times)$ is the product type former in $\mathsf{U_{SOP}}$ and $\mathsf{inject}_i : \mathsf{El_P\ A}_i \to \mathsf{El_{SOP}\ A}$. This is similar to the definition of non-dependent products in $\mathsf{U_{SOP}}$, except that we have to get rid of the type dependency by instantiating B with looping 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 $\mathsf{IsSOP}\ A$ and $\{a : A\} \to \mathsf{IsSOP\ (B\ a)}\}$, we define Rep for $\Sigma\ \mathsf{A\ B}$ as follows:

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

Then, the definition of encoding for Σ is only well-typed up to the fact that $\mathsf{rep}^{-1} \circ \mathsf{rep} = \mathsf{id}$:

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

This is, in fact, our reason for including the isomorphism equations as well 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 do not detail 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, this operation should be avoided when possible, since it has two states and can contribute to a blow-up of state size in combination with state-multiplying operations such as zip or concatMap. Both let-insertion and case splitting could be defined generically from single, concatMap and the following operation (definition omitted):

```
mapGen : (A → Gen B) → Pull A → Pull B
```

However, single would introduce superfluous states that way. Instead, we give more efficient specialized definitions. Let-insertion is as follows:

```
gen_Pull : {A : ValTy}{B : MetaTy} → ⇑A → (⇑A → Pull B) → Pull B
gen_Pull a f = Pull (Σ (⇑ A) (projS ∘ f)) ((a,) <$> projSeed (f a)) $ λ case
  Stop      → return Stop
  Skip s    → return $ Skip (a, s)
  Yield b s → return $ Yield b (a, s)
```

For case splitting, we have:

```
case_Pull : (Split A, IsSOP (SplitTo {A})) ⇒ ⇑A → (SplitTo {A} → Pull B) → Pull B
case_Pull a f = Pull (Σ (SplitTo {A}) (projS ∘ f))
                     (do {a′ ← split a; 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)
```

4.5 Discussion

Our stream library has a fairly 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 the 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" [**?**].

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

The Agda implementation follows this section very closely, so we do not detail it.

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,[2] and then define a case splitting function separately for each type. In each of these functions, we only need to deal with a concrete finite number of different machine state types, which is feasible with weak existentials. For example, we define

$$\mathsf{split}_{\mathsf{Pull}} : ⇑(\mathsf{List}\,A) \to (\mathsf{Maybe}_{\mathsf{M}}\,(⇑A, ⇑(\mathsf{List}\,A)) \to \mathsf{Pull}\,B) \to \mathsf{Pull}\,B.$$

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, when users of the library violate generativity.

REFERENCES

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