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

ANONYMOUS AUTHOR(S)

There are many abstraction tools in modern functional programming which heavily rely 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. In this paper we propose using a two-stage language to simultaneously get strong code generation guarantees and strong abstraction features. The object language is a simply typed first-order language where all function calls are statically known, and 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 develop some abstraction tools in this setting. First, we develop monads and monad transformers. Second, we develop fusion for push and pull streams. Most of our results are also adapted to a proof-of-concept library in typed Template Haskell.

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), 16 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 snippet.

$$\begin{aligned} &f :: \mathsf{Int} \to \mathsf{Reader}\ \mathsf{Bool}\ \mathsf{Int} \\ &f\,x = \textbf{do}\ \{b \leftarrow \mathsf{ask};\ \textbf{if}\ x\ \textbf{then return}\ (x+10)\ \textbf{else return}\ (x+20)\} \end{aligned}$$

With optimizations enabled, GHC compiles this roughly to the following:

$$\begin{aligned} &f :: \mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int} \\ &f = \lambda\,x\,b.\ \textbf{if}\ b\ \textbf{then}\ x+10\ \textbf{else}\ x+20 \end{aligned}$$

Without optimizations we roughly get:

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

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. The mapM function in the Haskell Prelude, which maps over a list in some monad, is a third-order and second-rank polymorphic function in disguise, because its monad dictionary argument contains the polymorphic second-order $(\gg\!=)$ method.

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

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.

REORG intro, trim, contrib, motivate monadtrans & stream fusion

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 sophisticated usage requires knowledge of GHC internals. For example, correctly specifying the *ordering* of certain rule applications is often needed. We have to also care about 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 — as explained in a source comment right next to the definition [**?**].

Although there are numerous tricks and idioms that are used in high-performance Haskell programming, for reliable performance it is necessary that programmers periodically *review* GHC's optimized code output. Needless to say, this is rather time-consuming and poorly scalable.

1.1 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 reliable. 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:

- Code generation might fail *too late* in the pipeline, producing incomprehensible errors; this is often caused by not having enough static guarantees about the well-formedness of code output.
- Tooling for the object language (debugging, profiling, IDE support) often does not work for metaprogramming. This is more likely if the object and meta layers are wildly different.

- Metaprogramming may introduce heavy noise and boilerplate, obscuring the logic of programs, or impose restrictions on how code can be structured. For instance, Template Haskell mandates that metaprograms used in splices in some module are defined in a different module.

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 downstream in the 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.

In particular, 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 locations. 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.

It turns out that 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. In other cases, the use of closures can be eliminated by small-scale defunctionalization. For example, difference lists [?] are often implemented with closures, but they can be also implemented as binary trees, with the same programming interface.

Essential closures: CPS? Threaded interpreters?

Whole-program defunctionalization is also possible, and it is notably used by the MLton compiler [?]. While this can be practical and effective, it can be also expensive and require whole-program processing. Also, conceptually speaking, it does not eliminate closures but instead makes them more transparent to optimizations. In this paper we do not use defunctionalization.

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 quite interesting to see how far we can go with it.

1.2 Contributions

CONTRIBUTIONS

2 BASICS OF CFTT

In the following we give an overview of CFTT features. Here we focus on examples and informal explanations. We defer the more formal details to Sections ?? and ??. 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 have nothing to do with staged compilation; they are about sizing for the purpose of logical consistency.

If we won't use Sigma notation, get rid of it

We use Agda-like syntax for functions and implicit arguments. A basic example:

$$\begin{aligned} &id : \{A : \mathsf{MetaTy}\} \to A \to A \\ &id = \lambda\, x.\, x \end{aligned}$$

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. Similarly, we can introduce implicit lambdas explicitly:

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

We write $(a : A) \times B$ for Σ-types in MetaTy. For the field projections, for $t : (a : A) \times B$, we have $\mathsf{fst}\, t : A$ and $\mathsf{snd}\, t : B[a \mapsto \mathsf{fst}\, t]$, and pairing is written simply as $(t,\, u)$. Additionally, we use syntactic sugar for field projections: we can write a type as $(field_1 : A) \times (field_2 : B) \times (field_3 : C)$, and write $t.field_2$ for a named field projection. The type itself denotes a right-nested Σ-type. We can also write $(t,\, u,\, v)$ for a right-nested iterated pairing. In Haskell style, we write () for the unit type, and also for its inhabitant.

Inductive types can be introduced using a Haskell-like ADT notation, or with a GADT-style one:

$$\textbf{data}\ \mathsf{Bool_M} : \mathsf{MetaTy} = \mathsf{True_M} \mid \mathsf{False_M} \qquad\qquad \begin{aligned} &\textbf{data}\ \mathsf{Bool_M} : \mathsf{MetaTy}\ \textbf{where} \\ &\quad \mathsf{True_M}\ \ : \mathsf{Bool_M} \\ &\quad \mathsf{False_M} : \mathsf{Bool_M} \end{aligned}$$

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 will also use Haskell-like newtype notation, such as in **newtype** Wrap A = Wrap $\{unWrap : A\}$.

Importantly, we can only produce meta-level values by induction on meta-level data. In general, all construction and elimination rules for type formers in MetaTy stay within MetaTy.

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 informally it is nicer to have an implicit subtyping.

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:

$$\begin{aligned}
&\mathit{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{add} := \mathbf{letrec}\,\mathit{go}\,n\,m := \mathbf{case}\,n\,\mathbf{of}\\
&\qquad\qquad \mathsf{Zero} \;\to m;\\
&\qquad\qquad \mathsf{Suc}\,n \to \mathsf{Suc}\,(\mathit{go}\,n\,m);\\
&\qquad \mathit{go}
\end{aligned}$$

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. Later on, we may use some implicit CFTT notations, but we will always disambiguate the stages of **let**-s in this way. Non-recursive **let** is also allowed, and can be used to shadow binders:

$$\begin{aligned}
&f : \mathsf{Nat} \to \mathsf{Nat}\\
&f\,x := \mathbf{let}\,x := x + 10; \mathbf{let}\,x := x + 20; x * 10
\end{aligned}$$

We also allow **newtype** definitions, both in ValTy and CompTy. These are assumed to be erased at runtime. In the Haskell adaptation they are important for aiding 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:

$$\begin{aligned}
&f : \mathsf{Bool} \to \mathsf{Nat} \to \mathsf{Nat}\\
&f\,b := \mathbf{case}\,b\,\mathbf{of}\,\mathsf{True} \to (\lambda\,x.\,x + 10); \mathsf{False} \to (\lambda\,x.\,x * 10)
\end{aligned}$$

What about **definitional equality** at the object level? This is a distinct notion from the runtime semantics; object programs are embedded in CFTT, which is dependently typed, and we need to decide definitional equalities 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 reasons are the following. First, since the object language has general recursion, most β-rules are only valid in the operational semantics up to some restrictions, and β-conversion is not decidable to begin with. Second, in metaprogramming we care about the size and efficiency of generated code, and these properties are not stable under β and η. Hence, the sensible choice is to do metaprogramming up to strict syntactic equality of object programs.

Let us discuss the object language. First, notice that there is no polymorphism or any kind of type dependency. Although we can define lists as **data** List A : ValTy := Nil | Cons A (List A), the parameterization is just a shorthand; all concrete instantiations of the type are distinct. This monomorphism makes it easy to use different memory layouts for different types. For example, types which look like products may be unboxed. We could also make a distinction between boxed and unboxed sum types. We do not explore this in detail, we just note that monomorphic types make it easy to control memory layouts, and we believe that this is an important part of performance optimization.

Second, there are no higher-order functions, and functions also cannot be stored in data structures. Hence, locally defined functions can never escape their scope, and all function calls are to functions defined in the current scope. This makes it possible to run object programs without using dynamic closures. This latter point might not be completely straightforward; what about the previous f function which has λ-expressions under a **case**, should that require closures?

We say that it should not. We make this formal formal in Section **??**; here we only give an intuitive explanation. In short, we choose a call-by-name runtime semantics for functions, which means that the only way we can compute with a function is by applying it to all arguments and extracting the resulting value. Hence, the only way to compute with f is to apply it to *two* arguments, so f is operationally equivalent to $\lambda\, b\, x.\, \mathbf{case}\; b\; \mathbf{of}\; \mathsf{True} \to x + 10; \mathsf{False} \to x * 10$. In Section **??** we show that all object programs can be transformed to a *saturated* form, where the arity of every function call matches the number of topmost λ-binders in the definition of the called function. Then, the local functions can be compiled either to top-level functions by lambda-lifting, or if they are only tail-called, left in place as join points [**?**].

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. The call-by-name nature of computation types requires a bit of a change of thinking from programmers, but we believe that it is well worth to have for the metaprogramming convenience.

On the other side of the spectrum, one might imagine generating object code in low-level A-normal form. This is more laborious, but it could be also interesting, because it forces us to invent abstractions for manipulating ANF in the metalanguage. In a similar vein, Allais recently proposed metaprogramming quantum circuits in a two-level type theory [**?**]. We leave such setups with low-level object languages to future investigation.

Finally, one might compare our object language to call-by-push-value (CBPV). Indeed, we took inspiration from CBPV, and there are similarities, but also differences. In both systems there is a value-computation distinction, and values are call-by-value, and computations are call-by-name. However, our object language allows variable binding at arbitrary types, while CBPV only supports it at value types. 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 work for us. Also, CBPV makes a judgment-level structural distinction between values and computations, while we use type universes for that. Generally speaking, type-based restrictions are easier to work with in dependent type theories than structural restrictions.

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 $⇑A : \mathsf{MetaTy}$, 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 $\langle t \rangle : ⇑A$, pronounced "quote t". This is the metaprogram which immediately returns t.
- For $t : ⇑A$, 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:

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

Here, *id* is used at type $⇑\mathsf{Nat}$. 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:

$$\begin{aligned} &and : ⇑\mathsf{Bool} \to ⇑\mathsf{Bool} \to ⇑\mathsf{Bool} \\ &and\, x\, y = \langle \mathbf{case} \sim x\ \mathbf{of}\ \mathsf{True} \to \sim y; \mathsf{False} \to \mathsf{False} \rangle \end{aligned}$$

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:

$$\begin{aligned} &map : \{A\, B : \mathsf{ValTy}\} \to (⇑A \to ⇑B) \to ⇑(\mathsf{List}\, A) \to ⇑(\mathsf{List}\, B) \\ &map\, f\, as = \langle \mathbf{letrec}\, go\, as := \mathbf{case}\, as\, \mathbf{of} \\ &\qquad\qquad\qquad \mathsf{Nil} \qquad\quad \to \mathsf{Nil}; \\ &\qquad\qquad\qquad \mathsf{Cons}\, a\, as \to \mathsf{Cons} \sim(f\, \langle a \rangle)\, (go\, as); \\ &\qquad\qquad go \sim as \rangle \end{aligned}$$

For example, this can be used as $\mathbf{let}\, f\, as : \mathsf{List}\,\mathsf{Nat} \to \mathsf{List}\,\mathsf{Nat} := \sim(map\, (\lambda\, x. \langle \sim x + 10 \rangle)\, \langle as \rangle)$. This is staged to a recursive definition where the mapping function is inlined into the Cons case as $\mathsf{Cons}\, a\, as \to \mathsf{Cons}\, (a + 10)\, (go\, as)$. Note that *map* has to abstract over value types, since lists can only contain values, not functions. Also, the mapping function has type $⇑A \to ⇑B$, instead of $⇑(A \to B)$. 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 $f : ⇑(A \to B)$, and f is staged to $\langle \lambda\, x. t \rangle$, then $\sim f u$ is staged to an undesirable "administrative" β-redex $(\lambda\, x. t)\, u$.

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 practical Haskell programming, and they also introduce a great amount of abstraction and 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 up to the runtime semantics of object programs, and we can convert back and forth in CFTT:

$$\begin{array}{ll} \mathit{up} : \Uparrow(A \to B) \to \Uparrow A \to \Uparrow B & \quad \mathit{down} : (\Uparrow A \to \Uparrow B) \to \Uparrow(A \to B) \\ \mathit{up}\, f\, a = \langle \sim\! f \sim\! a \rangle & \quad \mathit{down}\, f = \langle \lambda\, a.\, \sim\!(f\, \langle a \rangle) \rangle \end{array}$$

We can not 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 control dependencies are unavoidable.

We use a Haskell-style type class for binding time improvements:

$$\begin{array}{l} \textbf{class}\ \mathsf{Improve}\ (A : \mathsf{Ty})\ (B : \mathsf{MetaTy})\ \textbf{where} \\ \quad \mathit{up} \quad\ : \Uparrow A \to B \\ \quad \mathit{down} : B \to \Uparrow A \\ \textbf{instance}\ \mathsf{Improve}\ (A \to B)\ (\Uparrow A \to \Uparrow B)\ \textbf{where}\ \dots \end{array}$$

We will use type classes in an informal way, without precisely specifying how they work. However, in the Haskell adaptation of this paper we do make crucial use of type classes. There are some necessary differences between the CFTT and Haskell versions though, which we will summarize in Section **??**. Let us look at improvement for product types now:

$$\begin{array}{l} \textbf{instance}\ \mathsf{Improve}\ (A,\ B)\ (\Uparrow A,\ \Uparrow B)\ \textbf{where} \\ \quad \mathit{up}\, x \qquad\quad = (\langle \mathsf{fst} \sim\! x \rangle,\ \langle \mathsf{snd} \sim\! x \rangle) \\ \quad \mathit{down}\, (a,\ b) = \langle (\sim\! a,\ \sim\! b) \rangle \end{array}$$

Here we overload Haskell-style product type notation for types and pairing, at both levels (products are definable as a value type at the object level). There is a problem with this conversion though: *up* uses $x : \Uparrow(A,\ B)$ twice, which can increase code size and duplicate runtime computations. For example, $\mathit{down}\, (\mathit{up}\, \langle f\, x \rangle)$ is staged to $\langle (\mathsf{fst}\, (f\, x),\ \mathsf{snd}\, (f\, x)) \rangle$. 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. But it is impossible to use let-insertion in *up* because the return type is a product in MetaTy, and we cannot introduce object binders in meta-level code.

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.

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

This is a monad in MetaTy in a very standard sense, and we reuse Haskell type classes and **do**-notation.

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

From the Monad instance, the Functor and Monad instances can be also derived. We will 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 and returns an A value that can depend on produced object binders. Gen can be only "run" when the result type is an object type:

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

We can let-bind object expressions in Gen:

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

And also recursive function definitions:

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

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

$$\begin{aligned}&\textbf{instance}\ \mathsf{Improve}\ (A,\ B)\ (\mathsf{Gen}\ ({\Uparrow}A,\ {\Uparrow}B))\ \textbf{where}\\ &\quad \mathit{up}\ x \quad\ = \textbf{do}\ \{x \leftarrow \mathit{gen}\ x; \textbf{return}\ (\langle \mathsf{fst}\ {\sim}x\rangle,\ \langle \mathsf{snd}\ {\sim}x\rangle)\}\\ &\quad \mathit{down}\ x = \mathit{runGen}\ \$\ \textbf{do}\ \{(a,\ b) \leftarrow x; \textbf{return}\ \langle({\sim}a,\ {\sim}b)\rangle\}\end{aligned}$$

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? We may not want to do that, because we would like to reason about the size and makeup of the code we are generating, and the implicit code generation in Gen is not very transparent. 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:

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

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. Assume that Maybe_M is a meta-level monad, and MaybeT_M is the monad transformer with the standard definition:

$$\mathbf{newtype}\ \mathsf{MaybeT_M}\ M\ A = \mathsf{MaybeT_M}\ \{\mathit{runMaybeT_M} : M\ (\mathsf{Maybe_M}\ A)\}$$

Then the object-level Maybe is binding-time improved as follows:

$$\begin{aligned}
&\mathbf{instance}\ \mathsf{Improve}\ (\mathsf{Maybe}\ A)\ (\mathsf{MaybeT_M}\ \mathsf{Gen}\ ({\Uparrow} A))\ \mathbf{where}\\
&\quad \mathit{up}\ x = \mathsf{MaybeT_M}\ \$\ \mathsf{Gen}\ \$\ \lambda\, k.\\
&\qquad \langle \mathbf{case}\ {\sim} x\ \mathbf{of}\ \mathsf{Nothing} \to {\sim}(k\ \mathsf{Nothing_M}); \mathsf{Just}\ a \to {\sim}(k\ (\mathsf{Just_M} \langle a \rangle))\rangle\\
&\quad \mathit{down}\ (\mathsf{MaybeT_M}\ (\mathsf{Gen}\ \mathit{ma})) =\\
&\qquad \mathit{ma}\ (\lambda\, x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathsf{Nothing_M} \to \langle \mathsf{Nothing}\rangle; \mathsf{Just_M}\ a \to \langle \mathsf{Just}\ {\sim} a\rangle)
\end{aligned}$$

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

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

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

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

Assume also a $\mathit{lift} : \mathsf{Monad}\ M \Rightarrow M\ A \to \mathsf{MaybeT_M}\ M\ A$ operation, as in Haskell. We shall assume this for all monad transformers henceforth. We can do let-insertion in MaybeT_M Gen by simply lifting:

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

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:

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

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

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

3.3.1 *Case splitting in monads.*

Trying to write more substantial code, we quickly bump into an issue. We want to case-split on object-level data inside a monadic action, like in the following:

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

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:

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

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

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

Now, in any MonadGen, we may write

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

This can be generalized to any splitting on object values. In the supplemental Haskell implementation, we overload *split* with a type class. However, in a native implementation of CFTT it might make sense to simply extend **do**-notation with an elaboration of **case** on $\Uparrow A$-typed data to an application of the appropriate *split* function, so we can simply write a **case** *as* **of** ... in a **do**-block. We adopt this in the rest of this paper.

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:

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

We also switch from the object-level Maybe to a "transformer" form as well, and recover it as MaybeT Identity.

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

Generally, we want to improve an operator $F : \mathsf{ValTy} \to \mathsf{ValTy}$ to some $M : \mathsf{MetaTy} \to \mathsf{MetaTy}$, where M is a MonadGen. We define a *quantified class constraint* [?] for this, that we will often use as an assumption:

$$\mathsf{ImproveF}\ F\ M = \mathsf{MonadGen}\ M \times ((A : \mathsf{ValTy}) \to \mathsf{Improve}\ (F\ A)\ (M\ (\Uparrow A)))$$

With this, improvement can be generally defined for MaybeT:

$$\begin{aligned}&\textbf{instance}\ \mathsf{ImproveF}\ F\ M \Rightarrow \mathsf{Improve}\ (\mathsf{MaybeT}\ F\ A)\ (\mathsf{MaybeT_M}\ M\ (\Uparrow A))\ \textbf{where}\\ &\quad \mathit{up}\ x = \mathsf{MaybeT_M}\ \$\ \textbf{do}\\ &\qquad \mathit{ma} \leftarrow \mathit{up}\ \langle \mathit{runMaybeT} \sim\! x\rangle\\ &\qquad \textbf{case}\ \mathit{ma}\ \textbf{of}\ \mathsf{Nothing} \to \textbf{return}\ \mathsf{Nothing_M}\\ &\qquad\qquad\qquad\ \ \mathsf{Just}\ a \quad\ \to \textbf{return}\ (\mathsf{Just_M}\ a)\\ &\quad \mathit{down}\ (\mathsf{MaybeT_M}\ x) = \langle \mathsf{MaybeT} \sim\!(\mathit{down}\ \$\ x \gg\!= \lambda\, x.\ \textbf{case}\ x\ \textbf{of}\\ &\qquad \mathsf{Nothing_M} \to \textbf{return}\ \langle \mathsf{Nothing}\rangle\\ &\qquad \mathsf{Just_M}\ a \quad\ \to \textbf{return}\ \langle \mathsf{Just} \sim\! a\rangle))\rangle\end{aligned}$$

In the **case** in *up*, we already use our syntax sugar for matching on a Maybe inside an M action. This is legal, since we know from the ImproveF $F\ M$ assumption that M is a MonadGen.

In the meta level, we can reuse essentially all definitions from Haskell's monad transformer library mtl. The additional work that we have to do in the CFTT setting:

- Adding MonadGen instances, but these are trivially defined as $\mathit{liftGen} = \mathit{lift} \circ \mathit{liftGen}$ everywhere.
- Defining the object-level counterparts of monad transformers, and also the Improve instances.

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 $(S : \mathsf{ValTy})(F : \mathsf{ValTy} \to \mathsf{ValTy})(A : \mathsf{ValTy}) \to \mathsf{ValTy}$; the state parameter S has to be a value type, since it is an input to an object-level function.

```
instance $\mathsf{ImproveF}\ F\ M \Rightarrow \mathsf{Improve}\ (\mathsf{StateT}\ S\ F\ A)\ (\mathsf{StateT_M}\ (\Uparrow S)\ M\ (\Uparrow A))$ where
  $up\,x = \mathsf{StateT_M}\ \$\ \lambda\, s.$ do
    $as \leftarrow up\ \langle runStateT\ {\sim}x\ {\sim}s\rangle$
    case $as$ of $(a,\ s) \to$ return $(a,\ s)$
  $down\,x = \langle \mathsf{StateT}\ (\lambda\, s.\ {\sim}(down\ \$$ do
    $(a,\ s) \leftarrow runStateT_\mathsf{M}\ x\ \langle s\rangle$
    return $\langle({\sim}a,\ {\sim}s)\rangle)))\rangle$
```

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 $\mathsf{ImproveF}\ F\ M \Rightarrow \mathsf{Improve}\ (\mathsf{ReaderT}\ R\ F\ A)\ (\mathsf{ReaderT_M}\ (\Uparrow R)\ M\ (\Uparrow A))$ where
  $up\,x = \mathsf{ReaderT_M}\ \$\ \lambda\, r.\ up\ \langle runReaderT\ {\sim}x\ {\sim}r\rangle$
  $down\,x = \langle \mathsf{ReaderT}\ (\lambda\, r.\ {\sim}(down\ (runReaderT_M\ x\ \langle r\rangle)))\rangle$
```

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

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

is staged to

$$\langle \lambda\, x.\ ((),\ (x + x) + (x + x))\rangle$$

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 $\Uparrow()$ from actions instead of $()$ — the former is more convenient, because it can be immediately lowered with *down*.

```
$put' : (\mathsf{MonadState}\ (\Uparrow S)\ M,\ \mathsf{MonadGen}\ M) \Rightarrow\ \Uparrow S \to M\ (\Uparrow())$
$put'\ s =$ do $\{s \leftarrow gen\ s;\ put\ s;$ return $\langle()\rangle\}$

$modify' : (\mathsf{MonadState}\ (\Uparrow S)\ M,\ \mathsf{MonadGen}\ M) \Rightarrow (\Uparrow S \to\ \Uparrow S) \to M\ (\Uparrow())$
$modify'\ f =$ do $\{s \leftarrow get;\ put'\ (f\ s)\}$
```

$$\begin{aligned}&\mathit{local}' : (\mathsf{MonadReader}\,(\Uparrow R)\,M,\ \mathsf{MonadGen}\,M) \Rightarrow (\Uparrow R \to \Uparrow R) \to M\,A \to M\,A\\ &\mathit{local}'\ f\ \mathit{ma} = \mathbf{do}\,\{r \leftarrow \mathit{ask}; r \leftarrow \mathit{gen}\,(f\ r); \mathit{local}\,(\lambda\,\_.\,r)\ \mathit{ma}\}\end{aligned}$$

Now,

$$\mathit{down}\,\$\,\mathbf{do}\,\{\mathit{modify}'\ (\lambda\,x.\,\langle{\sim}x + {\sim}x\rangle); \mathit{modify}'\ (\lambda\,x.\,\langle{\sim}x + {\sim}x\rangle)\}$$

is staged to

$$\langle\lambda\,x.\,\mathbf{let}\,x := x + x; \mathbf{let}\,x := x + x; ((),\ x)\rangle$$

3.5 Joining Control Flow in Monads

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

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

The $\mathit{modify}'(\lambda\,x.\langle{\sim}x + 10\rangle)$ 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 two 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**. However, while this solution is fairly good in a state monad, where we can reasonably expect that product types can be unboxed by the object compiler, it introduces unnecessary runtime constructors for Maybe and other monads containing sum types. 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, we would like to produce object code for a **case** 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 finite meta-level sum of value types. Recall that Gen A is defined as $\{R : \mathsf{ValTy}\} \to (A \to \Uparrow R) \to \Uparrow R$. Here, the $A \to \Uparrow R$ continuation represents all possible code points where we can return to, and if A is a finite sum, we can rearrange $A \to \Uparrow R$ to a finite product of functions. Hence, we need a modest amount of generic programming with sums of values and products of computations.

$$\begin{aligned}&\textbf{data}\ \mathsf{SumVS} : \mathsf{List\ ValTy} \to \mathsf{MetaTy}\ \textbf{where} \ldots\\ &\textbf{data}\ \mathsf{ProdCS} : \mathsf{List\ CompTy} \to \mathsf{MetaTy}\ \textbf{where} \ldots\end{aligned}$$

We use a class for types that have a SumVS representation:

$$\begin{aligned}&\textbf{class}\ \mathsf{IsSumVS}\ (A : \mathsf{MetaTy})\ \textbf{where}\\ &\quad \mathit{Rep} \quad\ \ : \mathsf{List\ ValTy}\\ &\quad \mathit{encode} : A \to \mathsf{SumVS}\ \mathit{Rep}\\ &\quad \mathit{decode} : \mathsf{SumVS}\ \mathit{Rep} \to A\end{aligned}$$

We also define the mentioned isomorphism for Gen continuations.

$$\begin{aligned}&\mathit{tabulate} : (\mathsf{SumVS}\ \mathit{As} \to \Uparrow B) \to \mathsf{ProdCS}\ (\mathit{map}\ (\lambda\ A.\, A \to B)\ \mathit{As})\\ &\mathit{index} \quad\ \ : \mathsf{ProdCS}\ (\mathit{map}\ (\lambda\ A.\, A \to B)\ \mathit{As}) \to (\mathsf{SumVS}\ \mathit{As} \to \Uparrow B)\end{aligned}$$

We omit the definitions of *tabulate*, *index* and the IsSumVS instances, since they amount to standard generic programming; see e.g. [?]. The last basic definition we need is to let-bind all computations in a generic product:

$$\begin{aligned}&\mathit{genProdCS} : \mathsf{ProdCS}\ \mathit{As} \to \mathsf{Gen}\ (\mathsf{ProdCS}\ \mathit{As})\\ &\mathit{genProdCS}\ \mathsf{Nil} \qquad\quad\ \ = \textbf{return}\ \mathsf{Nil}\\ &\mathit{genProdCS}\ (\mathsf{Cons}\ c\ \mathit{cs}) = \mathsf{Cons} <\$> \mathit{gen}\ c <*> \mathit{genProdCS}\ \mathit{cs}\end{aligned}$$

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

$$\textbf{class}\ \mathsf{MonadJoin}\ M\ \textbf{where}\ \mathit{join} : \mathsf{IsSumVS}\ A \Rightarrow M\ A \to M\ A$$

The most interesting instance is for Gen.

$$\begin{aligned}&\textbf{instance}\ \mathsf{MonadJoin\ Gen}\ \textbf{where}\\ &\quad \mathit{join}\ \mathit{ma} = \mathsf{Gen}\ \$\ \lambda\, k.\ \mathit{runGen}\ \$\ \textbf{do}\\ &\qquad \mathit{joinpoints} \leftarrow \mathit{genProdCS}\ (\mathit{tabulate}\ (k \circ \mathit{decode}))\\ &\qquad a \leftarrow \mathit{ma}\\ &\qquad \textbf{return}\ \$\ \mathit{index}\ \mathit{joinpoints}\ (\mathit{encode}\ a)\end{aligned}$$

Here we first convert $k : A \to \Uparrow R$ to a product of join points and let-bind each of them. Then we adjust the return code points of *ma* by looking up the corresponding continuation from the list of join points, using *index*. We also need MonadJoin instances for the other transformers in our library, but these are all essentially just lifting *join*. In $\mathsf{StateT_M}$ we must have IsSumVS for S because S appears in the transformer stack's return value.

$$\begin{aligned}&\textbf{instance}\ (\mathsf{MonadJoin}\ M) \Rightarrow \mathsf{MonadJoin}\ (\mathsf{MaybeT_M}\ M)\ \textbf{where}\\ &\quad \mathit{join}\ (\mathsf{MaybeT_M}\ \mathit{ma}) = \mathsf{MaybeT_M}\ (\mathit{join}\ \mathit{ma})\\ &\textbf{instance}\ (\mathsf{MonadJoin}\ M) \Rightarrow \mathsf{MonadJoin}\ (\mathsf{ReaderT_M}\ R\ M)\ \textbf{where}\\ &\quad \mathit{join}\ (\mathsf{ReaderT_M}\ \mathit{ma}) = \mathsf{ReaderT_M}\ (\mathit{join} \circ \mathit{ma})\\ &\textbf{instance}\ (\mathsf{MonadJoin}\ M,\ \mathsf{IsSumVS}\ S) \Rightarrow \mathsf{MonadJoin}\ (\mathsf{StateT_M}\ S\ M)\ \textbf{where}\\ &\quad \mathit{join}\ (\mathsf{StateT_M}\ \mathit{ma}) = \mathsf{StateT_M}\ (\mathit{join} \circ \mathit{ma})\end{aligned}$$

Now, whenever the return value of a **case**-splitting action is a sum 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 block inlining into the branches, without creating any runtime sum types.

bigger example

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 quite liberally pattern match on object-level values in monadic code, insert object-level **let**-s with *gen* and control code size by *join*-s.
- 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*. Everything else is 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 quite a bit of 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 make the concession that stages of let-definitions are always specified (as we do here). Moreover, *up* and *down* should be also effectively inferable, using bidirectional elaboration, but it would make sense to limit inferred binding-time improvement to monads only. With stage inference and binding-time improvement inference, code in CFTT would look only modestly more complicated than in Haskell, and we would get far more robust and controllable code generation.

Second, in CFTT we cannot store most monadic actions in runtime data structures, those that are implemented using functions, nor can we have functions in State state, or in Reader environments. However:

- Essential usage of this is rare in practice.
- It is possible to extend CFTT with a closure type former, converting computations to values, in which case there is no such limitation anymore. Here, closure-freedom is still available; we can choose to use or abstain from the explicit closure type.

3.7 Haskell Implementation

We implemented all features in this section in typed Template Haskell [?], using only the strongly typed interface. The organization of the code is essentially the same as here, but some changes are necessary:

- TH cannot properly handle scoping of type variables across quotations, generating ill-typed code in some cases. It is fairly easy to work around this using term-level proxy values.
- We do not bother tracking value and computation types. This could be done, using dummy type classes, but we would not get any extra guarantees from it in downstream compilation.
- When we take the product of generic sums in CFTT, we simply take the object-level product of component types, pairwise. We do this because we assume that the downstream compiler can represent products without overheads, in an unboxed manner, and also erase unit types. In Haskell we do not rely on this, and instead use a slightly more complicated sum-of-products-of-values representation, and also fuse away product types during staging.
- We make heavy use of *singleton types* [?] in generic sums-of-products, to emulate dependent types. This adds significant noise compared to a native dependently typed version.

- We have two separate type classes for binding-time improvement, one for base types and one for monads specifically. This is because the quantified constraints in **??** yield poor type inference for *up* and *down*, by preventing *functional dependencies* [?] between the class parameters. We expect that in a native CFTT implementation, the extra staging information in type universes would help inference in this case.

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. For example, mapping twice over a list produces an intermediate list in a naive implementation but does not do so in a fused implementation.

Fusion techniques 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, which manifests as *state machines*. 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 fusion.

The reason is partly lack of space, but also that pull fusion has been historically the more difficult to reliably compile, and we can demonstrate significant improvements in CFTT, in robustness and expressive power. Our fusion always goes through, and we also support a *concatMap* operation, which has been elusive in guaranteed fusion. We believe that our implementation also provides valuable conceptual clarity.

4.1 Streams

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

$$\textbf{data}\ \mathsf{Step}\ S\ A = \mathsf{Stop} \mid \mathsf{Yield}\ A\ S \mid \mathsf{Skip}\ S$$

$$\textbf{data}\ \mathsf{Pull}\ (A : \mathsf{MetaTy}) : \mathsf{MetaTy}\ \textbf{where}$$

$$\quad \mathsf{MkPull} : (S : \mathsf{MetaTy}) \to \mathsf{IsSumVS}\ S \Rightarrow S \to (S \to \mathsf{Gen}\ (\mathsf{Step}\ S\ A)) \to \mathsf{Pull}\ A$$

In MkPull, S is the type of the internal state which is required to be a finite sum of values by the $\mathsf{IsSumVS}\ S$ constraint. The next S-typed field is the initial state, while $S \to \mathsf{Gen}\ (\mathsf{Step}\ S\ A)$ represents all transitions. 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).

5 TODOS

Introduce newtypes accordingly

allow arbitrary parameters (including ValTy -> ValTy functions) for object ADTs

extend as I have more stuff

FIGURE OUT: monadic tailrec

FIGURE OUT: Traversals

DEVELOP: mutual letrec based on computational product types

REFERENCES