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), 8 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.

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

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

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

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

Without optimizations we roughly get:

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

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 $\lambda b.$ ... 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.

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

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.

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 staging in the sense of staged compilation; they are about sizing for the purpose of logical consistency.

We use Agda-style syntax and implicit arguments. A basic example:

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

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. 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:

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

$$\begin{array}{rl} \multicolumn{2}{l}{\textbf{data}\ \mathsf{Bool}_\mathsf{M} : \mathsf{MetaTy}\ \textbf{where}} \\ \mathsf{True} & : \mathsf{Bool}_\mathsf{M} \\ \mathsf{False} & : \mathsf{Bool}_\mathsf{M} \end{array}$$

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 use this subscript to distinguish the meta-level version.

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. It is further split to two sub-universes.

First, **ValTy** : MetaTy is the universe of *value types*. ValTy supports parameterized algebraic data types, with two restrictions:

- ADTs can be only parameterized over types in ValTy.
- Constructor fields 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 domain is a value type:

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

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. An example for an object-level program, where we already have natural numbers declared as **data** Nat : ValTy := Zero | Suc Nat:

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

Every recursive definition must be introduced with **letrec** . Note also the single semicolon after the **case** branches. The general syntax is **letrec** $x := t; u$, optionally with a type annotation for x on a separate line.

Also note that object-level definitions use := instead of = for meta-level ones. Generally, let-definitions are the biggest source of stage ambiguity in surface notation. Later on, we will use more implicit staging notation, 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:

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

let-definitions can be used to define values of any type, and the type of the **let** body can be also anything. Additionally, the right hand sides of **case** branches can also have arbitrary types. So the following is well-formed:

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

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 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 the following definition:

$$\lambda\, b\, x.\, \textbf{case}\ b\ \textbf{of}\ \mathsf{True} \longrightarrow x + 10; \mathsf{False} \longrightarrow x * 10$$

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 function definition.

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 for the object language. 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 choose to 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 it 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 in a closure value, 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.

3 TODOS

extend as I have more stuff

FIGURE OUT: monadic tailrec

FIGURE OUT: Traversals

DEVELOP: mutual letrec based on computational product types

4 THE OBJECT LANGUAGE

Describe object language. Simple types + first-order fun + fixpoints + ADT-s.

Show Agda formalization basically, types and syntax.

Show Agda definitional interpreter. Discuss closure-free evaluation

Examples, properties. Maybe big-step reduction? Maybe program equivalence? Look at Harper PFPL for program equivalence reference.

Sketch downstream compilation.

- Eta-expand function definitions to arity.
- Turn functions in immediate beta-redexes to let-bindings.
- Lambda-lift non-tail-called functions to top.
- Keep only-tail-called local functions as join points.

Downstream compilation should erase the unit type and implement products as "flat" unboxed structures.

Discuss object language design choices.

- Can be more obviously first-order: e.g. no currying, only Val->Val functions, or ANF functions. This requires more bureaucracy in metaprogramming.
- Can be more flexible but less obviously first-order: allow function let bodies, allow function case bodies. Requires more translation. More convenient to program in. In future work.

5 THE STAGED LANGUAGE

Don't give horizontal-bar rules.

Meta-features:

- Pi, Sigma, Unit, Bottom, Universes (indices elided), ADT-s. Ind families probably not needed.
- Agda notation.
- Type classes informally in Haskell notation?

Explain staging ops, staging algorithm. Explain object embedding. Refer to 2LTT paper for correctness.

Mention stage annotation inference, but we'll not use it often! To make things explicit.

We should include a generous amount of staging examples, and some traced staging.

Basic staging examples: identity, map, power function. Boolean short-circuiting.

Basic staged classes, Eq, Ord, Show, Monoid, etc. Vectors with computed length.

6 PROGRAMMING WITH MONADS

6.1 Basic Binding Time Improvements

Notion. For functions. For products. Note that for products there is a computation duplication problem. Mention let-insertion.

6.2 The Code Generation Monad

Is fundamental to this work.

Definition. Explanation. Define Functor, Applicative, Monad instances. Define runGen, runGen1, runGenN etc. Define let-insertion.

Basic examples.

Explanation in terms of Church-coding.

Gen Vector mapping or folding example from 2LTT repo. Without Gen, it's a big pain to get ideal linear-size map code in N. With Gen it's easy-peasy.

6.3 Monad Transformers

Identity, ReaderT, StateT, MaybeT, ExceptT

Strict versions of put, modify, local, runner functions.

MTL-style overloading. MonadReader, MonadState, MonadError, MonadGen, liftGen

Binding time improvement for monads. Up/down overloading.

6.4 Case Splitting in MonadGen

It works for any object ADT.

Introduce sums-of-values (SOV).
Introduce syntax sugar for object case splits in MonadGen blocks.
Basic examples.

6.5 Joining Control Flow

The problem of join points. Motivation in terms of MaybeT Gen.

We can let-bind computations by tripping through up/down of monads, but that introduces unbox/rebox cost for Maybe & Either.

Works OK for Reader/State though.

```
f : Bool -> MaybeT0 Gen ^Int
f = \b. ~(down do
  x <- case <b> of
    True  -> gen <10>
    False -> gen <20>
  y <- gen <~x + 10>
  z <- gen <~x * 10>
  pure <~x + ~y + ~z>)
```

Join points using SOV-s. Description of SOV machinery. MonadJoin. instance MonadJoin Gen. MonadJoin MaybeT.

6.6 Monadic Tail Recursion

`replicateM_`, `find` functions working in "any" monad. Non-tail recursion is not possible to fuse, return addresses are dynamic. Tail recursion can return to statically known points.

6.7 Discussion

Limitations. We can't put non-value actions into structures, can't accumulate them.

Guarantees/properties. Closure-freedom. No administrative redexes. No boxing in join points. Only boxing in monadic code comes from "down". No code blowup.

7 FUSION

Motivation.

7.1 Push streams

Definition.

7.2 Pull streams

Pull streams are definitely not monadic. `Nat -> Stream a` contains "infinite" amount of code internally.

Pull streams are also not selective functors.

We do have monadic binding for lifted types, plus "case" matching on arbitrary object values! This is practically almost as good as having unlimited monadic bind! We need to assume a "non-dependency axiom", that any $f : \Uparrow A \to \mathsf{ValTy}$ must be constant.

Validate non-dependency in the psh model!

Add meta-identity type to the psh model

REFERENCES