Closure-Free Functional Programming in 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 as a *two-level type theory*. We develop abstraction tools in this setting and demonstrate two use cases. 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 Two-Level Type Theory. 1, 1 (January 2024), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

FIGURE OUT: monadic tailrec

FIGURE OUT: Traversable

DEVELOP: mutual letrec based on computational product types

Motivation.

Lots of abstraction in FP. Heavy reliance on general-purpose compiler optimization, without guarantee of code quality.

Closures introduce heap allocation and dynamic control flow. Compilers try hard to get rid of closures.

Here: use staging instead of closures. Essential use of closures is remarkably rare.

General-purpose opt is slow, fragile, but automatic.

Metaprog is fast, deterministic, robust but requires user control.

We make metaprog easier and more streamlined.

Defunctionalization. Full defunc requires whole-program compilation. Our approach does not use defunc.

ACM XXXX-XXXX/2024/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Apparent closure use-cases can be locally defunctionalized. DList prime example.
Closure use cases:

- threaded interp, closure interp
- Higher-order DSL-s (but they could be also staged to FO!)
- "Monadic reflection" in effects
- ListT working on any monad
- CPS

Summary of contributions, sections, incl Typed TH implementation.

2 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.

3 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.

4 PROGRAMMING WITH MONADS

4.1 Basic Binding Time Improvements

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

4.2 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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.

4.7 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.

5 FUSION

Motivation.

5.1 Push streams

Definition.

5.2 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 $f : \Uparrow A \to \mathrm{ValTy}$ must be constant.

Validate non-dependency in the psh model!

Add meta-identity type to the psh model

REFERENCES