# Catching fire: Undefined behaviour for equational reasoning of lazy languages with strictness annotations

Subtitle

Anonymous Author(s)

## Abstract

We propose a new style of semantics for non-strict languages that provide access to many optimizations that are currently unsound for lazy languages. Lazy languages, while providing equational reasoning, are unable to reason effectively about strictness annotations. On the other hand, most strict languages are impure, and thus unable to provide equational reasoning due to the presence of side effects. Research on optimizing non-strict languages has focused on optimizing away non-strictness with demands, and efficient lowering of strictness onto modern hardware. We take a different approach: we propose to use undefined behaviour judiciously while defining the semantics of our language to enable many key optimizations that are possible in the strict world, that are not possible in the non-strict world. We model divergent computations as undefined behaviour, giving the compiler far more freedom to optimize and reorder a mix of strict and non-strict code.

## 1 Introduction

Lazy languages such as Haskell offer many benefits, chief of which is the ability to reason equationally about programs. For performance, considerations, the prototypical lazy language, Haskell, provides "bang patterns" to mark strictness. Herein lies the trouble; these "bang patterns" do not permit equational reasoning for the user. Worse, they form a sequence point in the otherwise pure semantics, taking away most of the optimization freedom from the compiler. We propose to remedy the situation by making judicious use of undefined behaviour semantics. While undefined behaviour is often reviled in discourse, it is in fact a major reason why C and C++ can be optimized as well as they can; defining certain rare or impossible conditions as undefined behaviour provides the compiler a great deal of freedom by being able to ignore these cases. We choose to mark divergence as undefined behaviour. This solves the correctness problem in reordering strict computations; The only thing our compiler (Lizzy) will reason about is performance when reordering strict computations. Furthermore, this is not too insane a proposal; After all, the C standard also defines side-effect free infinite loops as undefined behaviour. Next, we show a variety of examples where this simple extension to the language semantics permits far easier reasoning about mixing strictness and non-strictness. We then put this to the test, by extending MLIR with a new non-strict IR, called `lz` . Note that the undefined behaviour semantics are critical in order to reason effectively about the mix of strict and non strict behaviour we wish to model within MLIR. Finally, to validate our approach, we consider a series of benchmarks where our programming language, Lizzy outperforms similar programs in Haskell compiled by the Glasgow Haskell Compiler.

Our contributions are:

- A new dialect (`lz`) for MLIR to encode these semantics, and provide modular non-strictness.
- A translator from STG to `lz` that exhibits the ability to eliminate laziness and leverage MLIR's loop optimisation framework.
- A rewrite driven strategy for worker/wrapper that interleaves worker/wrapper analysis along with worker/wrapper rewriting, by phrasing the worker/wrapper problem as an outlining/inlining problem.

## 2 Stuff that doesn't work

- circular references: SSA by default cannot handle circular references. We need to use the "relaxed SSA" that allows graphs. It's unclear how well this adapts.
- A garbage collector. The usual story here, we don't really make a fair comparison.
- Any realistic program exhibiting "difficult" laziness. I feel what would be interesting would be to implement, say, GRIN, along with a sophisticated pointer analysis using the MLIR analysis-as-rewrite story here.
- A proof of correctness of the naive demand analysis I wrote.

## 3 Stuff that does work

- An encoding of laziness that works for the simple stuff I've tried
- A simple version of "demand analysis" that eliminates first order laziness.
- Some toy examples where we generate better IR than GHC because we can expose the "real work" to LLVM/MLIR in a way that they understand, by eliminating the noise that is laziness.

## 4 What I feel would make a good paper/thesis

- Our story of performing worker/wrapper by outlining/inlining is intriguing. I don't think it's strong. So, to remedy that, we can:
- (1) Make our laziness encoding bulletproof, connect to CBPV for example, handle circular references, etc.
- (2) Make a strong case that MLIR is an interesting target for this breed of language. So, perform unification in MLIR, typeclass resolution in MLIR, perform some analysis/optimisation. Ths way, we show an end-to-end prototype of something that sorta works, instead of one thing that sorta works (?)

**TODO: fill in semantics here.**

## 5 `lz` : Its Syntax, Type system, semantics

### 5.1 A high level overview of `MLIR`

### 5.2 A high level overview of `lz`

We extend the basic `std` dialect modestly, by adding primitives that can describe non-strictness. We first introduce a primitive known as `lz.thunk` , which is responsible for creating non-strict thunks. To represent lazy function application, we introduce a new operation called `lz.ap` , which is the lazy sibling of `std.call` . This creates a thunk, which when evaluated, invokes the function with the given arguments. To force a thunk, we introduce `lz.force`, which receives a thunk as input and returns the forced value of the thunk as output. In effect, we have an encoding of administrative normal form as (ANF) an MLIR dialect.

To represent algebraic data types, which enable reasoning of control flow and are the primary mode of abstraction in most functional languages, as well as newer imperative languages such as Rust, we introduce the `lz.construct` and `case` operators, which are the introduction and elimination forms of structured data in the dialect.

**TODO: this is broken, because it doesn't scope properly. We need to describe `lz.thunk` as building a thunk in memory/on the heap, and the value stored in the register file is the pointer to this thunk**.

### 5.3 Purifying `lz.force`

Let us for a moment assume that divergence is not undefined behaviour. Now consider the code in **??**. The variable called `%loopv` is unused. However, because divergence `%loopv` is divergent, it is incorrect to eliminate the variable definition of `%loopv`, for the divergence of `loop` occurs when `%loopt` is forced; That is, when the instruction `%loopv = lz.force(%loopv)` is executed. This implies that the instruction `lz.force`is side-effecting.

If we wished to regain purity of `lz.force`and leverage the power of SSA, then we must be allowed to eliminate a call to `lz.force`if the value is unused. This is exactly the same requirement as being able to equationally reason with bangs in a `let` binding. In that use-case, we wished for undefined behaviour semantics for ease-of-reasoning. Here, we wish for undefined behaviour semantics for ease-of-optimization.

```
// loop = loop
lz.func @loop () -> !lz.adtBox> lz.return(// main =
    let !x = loop in 42lz.func @main() ->
    !lz.adt<@Int> : !lz.thunk<lz.adt<@Box>>:
    !lz.adt<@Box>lz.return(
```

**Listing 1.** The SSA encoding of a lazy program with divergence in the `lz` dialect. Note the unused divergent variable `%loopv`

```
// loop = loop
lz.func @loop () -> !lz.adtBox> lz.return(// main =
    let !x = loop in 42lz.func @main() ->
    !lz.adt<@Int> // // : !lz.thunk<lz.adt<@Box>>//
    // : !lz.adt<@Box>lz.return(
```

**Listing 2.** Under the assumption that divergence is UB we can eliminate the call to `lz.force`. This regains SSA semantics.

### 5.4 Purifying `lz.construct`

Consider the program in **??**. For us to eliminate the call to `lz.construct`, we need to be sure that memory allocation has no side effects. In our case, since we assume that allocations are not visible to the user in terms of their implementation details (eg. The address of the pointer of the block of memory), we can safely remove the value `%unused = lz.construct(@Box)` as `%unused` is unused.

On the flipside, consider a program where we want to copy the instruction `lz.construct`. If we were not guaranteed the existence of a garbage collector, then this would not be pure; we would have created a chunk of memory that is never freed. **TODO: Is there a good example where we want to copy a value for optimization? Something like, copy an SSA value both into a loop and outside or something?**

```
lz.func() -> !hask.adtBox> lz.ret(
```

**Listing 3.** An unused called to `lz.construct`

## 6 Worker wrapper by local rewrites: A first stab

In this section, we consider a small example program, which we will explain how to perform a classical transform (the worker-wrapper transform) purely by using local rewrites. Traditionally, such a transformation is performed by first performing a demand analysis whose results are used to drive the rewrite.

We explore how this transformation can be achieved without the need for demand analysis by performing local rewrites. This provides both a simpler manifestation of the algorithm,

```
toplevel := func | global
func := <fn-name> formal-param-list -> <ret-type> <region>
region := list [bb]
bb := <bb-name> formal-param-list : list inst; terminator-inst
inst := retval "=" <op-name> arg-list
terminator-inst := std.return(<name>) | br <name> | condbr <name> <bb1> <param-list> <bb2> <param-list>
```

**Figure 1.** The std dialect syntax

$$\frac{}{(\%\texttt{vid} = \texttt{std.constant}\ const, R) \xrightarrow{\texttt{asgn}} R[\%\texttt{vid} \mapsto const]}$$

$$\frac{\begin{array}{l} R[x_1] = v_1 \quad R[x_2] = v_2 \quad \ldots \quad R[x_n] = v_n \\ [\![\texttt{f}]\!]\,(v_1, v_2, \ldots, v_n) = y \end{array}}{(\%\texttt{vid} = \texttt{f}\,(x_1, x_2, \ldots, x_n), R) \xrightarrow{\texttt{asgn}} R[\%\texttt{vid} \mapsto y]}$$

$$\frac{\begin{array}{l} P[pc] = \texttt{condbr c \^{}bb1(xs) \^{}bb2(ys)} \\ P[\texttt{\^{}bb1}] = \texttt{\^{}bb1(ls)} \\ R[c] \neq 0 \end{array}}{(pc, R) \xrightarrow{\texttt{ctrl}} (pc', R[ls \mapsto R[xs])}$$

$$\frac{P[pc] = \mathsf{br}\ c\ l(v_1)\ r(v_2) \quad P[r] = pc' \quad R[c] = 0}{(pc, R) \xrightarrow{\texttt{ctrl}} (pc', R[r.input \mapsto R[v_2])}$$

$$\frac{P[pc] = \mathsf{br}\ n(v) \quad P[n] = pc'}{(pc, R) \xrightarrow{\texttt{ctrl}} (pc', R[n.input \mapsto R[v])}$$

**Figure 2.** The operational semantics of the std dialect

```
thunk(x): T -> Thunk<T>
ap(f, v1, v2, ..., vn): ((T1,...Tn) -> R) x T1 x ... Tn -> Thunk<R>
force(x): Thunk<T> -> T
construct(ConsName, v1, ..., vn): Symbol x T1 x T2 ... x Tn -> ADT<T>
case(x: T) of ... : ADT<T> -> R
```

**Figure 3.** Syntax of lz extensions

as well as a potentially faster implementation, as the MLIR infrastructure is capable of performing rewrites in parallel.

```
SimpleInt f(Thunk<SimpleInt> i) {
    SimpleInt icons = force(i);
    int ihash = casedefault(icons);
    if (ihash <= 0) {
        return SimpleInt(42);
    } else {
        int prev = ihash - 1;
        SimpleInt siprev = SimpleInt(prev);
        Thunk<SimpleInt> siprev_t = thunkify(siprev);
        SimpleInt f_prev_v = apStrict(f, siprev_t);
        return f_prev_v;
    }
}

int main() {
    printf("%d\n", f(thunkify(SimpleInt(1))).v);
}
```

**Listing 4.** initial source code

```
SimpleInt f2(SimpleInt icons) {
    int ihash = casedefault(icons);
    if (ihash <= 0) {
        return SimpleInt(42);
    } else {
        int prev = ihash - 1;
        SimpleInt siprev = SimpleInt(prev);
        Thunk<SimpleInt> siprev_t = thunkify(siprev);
        SimpleInt f_prev_v = f(siprev_t);
        return f_prev_v;
    }
```

$$\frac{\begin{array}{l} P[pc] = \\ \quad \texttt{\%vid = lz.construct(@ConsName, \%x1, ..., \%xn)} \\ P[\texttt{\%x1}] = v_1, \quad \ldots, \quad P[\texttt{\%xn}] = v_n \end{array}}{R[\texttt{\%vid} \mapsto \texttt{constructor(@ConsName, } v_1, \ldots, v_n\texttt{)}]}$$

$$\frac{\begin{array}{l} P[pc] = \texttt{\%vid = lz.ap(\%fref, \%x1, ..., \%xn)} \\ P[\texttt{\%f1}] = \texttt{ref(@fn)} \\ P[\texttt{\%x1}] = v_1, \quad \ldots, \quad P[\texttt{\%xn}] = v_n \end{array}}{R[\texttt{\%vid} \mapsto \texttt{thunk(@fn, } v_1, \ldots, v_n\texttt{)}]}$$

$$\frac{\begin{array}{l} P[pc] = \texttt{\%vid = lz.case(\%x, @Cons}_1\texttt{, r}_1\texttt{, ..., @Cons}_N\texttt{, r }_N\texttt{)} \\ \texttt{@Cons}_i = \texttt{@XCons} \\ P[\texttt{\%x}] = \texttt{construct(@XCons, } v_1, \ldots, v_n\texttt{)} \\ r_i = \texttt{ \{ \^{}entry(arg1, ..., argn): ... \}} \end{array}}{(pc, R) \xrightarrow{\texttt{ctrl}} (P[r_i], R[\texttt{arg}_i \mapsto v_i])}$$

$$\frac{\begin{array}{l} p[pc] = \texttt{\%vid = lz.force(\%thnk)} \\ p[\texttt{\%thnk}] = \texttt{thunk(@fn, } v_1, \ldots, v_n\texttt{)} \\ [\![\texttt{@fn}]\!](v_1, v_2, \ldots, v_n) = y \end{array}}{r[\texttt{\%vid} \mapsto y]}$$

**Figure 4.** Operational semantics of `lz` extensions

```
}
SimpleInt f(Thunk<SimpleInt> i) {
    SimpleInt icons = force(i);
    SimpleInt ret = f2(icons);
    return ret;
}
```

**Listing 5.** Step 1: outlining everything after the initial `lz.forceinto` f2

```
SimpleInt f2(SimpleInt icons) {
    int ihash = casedefault(icons);
    if (ihash <= 0) {
        return SimpleInt(42);
    } else {
        int prev = ihash - 1;
        SimpleInt siprev = SimpleInt(prev);
        Thunk<SimpleInt> siprev_t = thunkify(siprev);
        SimpleInt f_prev_v = f(siprev_t);
        Thunk<SimpleInt> f_prev_v = f2(siprev);
        return f_prev_v;
    }
}

SimpleInt f(Thunk<SimpleInt> i) {
    SimpleInt icons = force(i);
    SimpleInt ret = f2(icons);
    return ret;
}
```

**Listing 6.** Step 2: Removing the un-necessary lazy recursive call

```
SimpleInt f3(int icons) {
    if (ihash <= 0) {
        return SimpleInt(42);
    } else {
        int prev = ihash - 1;
        SimpleInt siprev = SimpleInt(prev);
        Thunk<SimpleInt> f_prev_v = f2(siprev);
        return f_prev_v;
    }
}

SimpleInt f2(SimpleInt icons) {
    int ihash = casedefault(icons);
     SimpleInt ret = f3(ihash);

    return ret;
}
...
```

**Listing 7.** Step 3: outline everything after casedefault into f3

```
SimpleInt f3(int icons) {
    if (ihash <= 0) {
        return SimpleInt(42);
    } else {
        int prev = ihash - 1;
         SimpleInt siprev = SimpleInt(prev);
         Thunk<SimpleInt> f_prev_v = f2(siprev);
         Thunk<SimpleInt> f_prev_v = f3(prev);
        return f_prev_v;
    }
}
...
```

**Listing 8.** replace call `f2(SimpleInt(prev))` to `f3(prev)`

```
SimpleInt f3(int icons) {
    if (ihash <= 0) {
        return SimpleInt(42);
    } else {
        int prev = ihash - 1;
         Thunk<SimpleInt> f_prev_v = f3(prev);
        return f_prev_v;
    }
```

```
}

SimpleInt f2(SimpleInt icons) {
    int ihash = casedefault(icons);
     SimpleInt ret = f3(ihash);
    return ret;
}
SimpleInt f(Thunk<SimpleInt> i) {
    SimpleInt icons = force(i);
    SimpleInt ret = f2(icons);
    return ret;
}
```

**Listing 9.** taking stock

At this point, we have a clean program that has been worker/wrappered. We can see that the call chain looks as follows:

$$f \xrightarrow{\texttt{force}} f_2 \xrightarrow{\texttt{unwrap}} f_3 \to f_3 \ldots \xrightarrow{\texttt{wrap}} f_2 \xrightarrow{\texttt{thunk}} f$$

Of course, much remains to be discussed: what about sum types? what about non-tail recursion? Extensions to the same idea will handle the above problems, while retaining the pleasing simplicity of this outlining/inlining paradigm.

## 7 Worker wrapper by local rewrites: Non-tail-calls

```
SimpleInt g(Thunk<SimpleInt> i) {
  SimpleInt icons = force(i);
  int ihash = casedefault(icons);
  if (ihash <= 0) {
    return SimpleInt(42);
  } else {
    int prev = ihash - 1;
    SimpleInt siprev = SimpleInt(prev);
    Thunk<SimpleInt> siprev_t = thunkify(siprev);
    SimpleInt g_prev_v = g(siprev_t);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
  }
}
```

**Listing 10.** Step 0: the initial program

```
SimpleInt g2(SimpleInt i) {
  int ihash = casedefault(icons);
  if (ihash <= 0) {
    return SimpleInt(42);
  } else {
    int prev = ihash - 1;
    SimpleInt siprev = SimpleInt(prev);
    Thunk<SimpleInt> siprev_t = thunkify(siprev);
    SimpleInt g_prev_v = g(siprev_t);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
  }
}
SimpleInt g(Thunk<SimpleInt> i) {
  SimpleInt icons = force(i);
  g2(icons);
}
```

**Listing 11.** Step 1: outline everything after `force`

```
SimpleInt g2(SimpleInt i) {
  int ihash = casedefault(icons);
  if (ihash <= 0) {
    return SimpleInt(42);
  } else {
    int prev = ihash - 1;
    SimpleInt siprev = SimpleInt(prev);
    Thunk<SimpleInt> siprev_t = thunkify(siprev);
    SimpleInt g_prev_v = g(siprev_t);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
  }
}
SimpleInt g(Thunk<SimpleInt> i) {
  SimpleInt icons = force(i);
  g2(icons);
}
```

**Listing 12.** Step 2: outline everything after `force`

```
SimpleInt g2(SimpleInt i) {
  int ihash = casedefault(icons);
  if (ihash <= 0) {
    return SimpleInt(42);
  } else {
    int prev = ihash - 1;
    SimpleInt siprev = SimpleInt(prev);
    Thunk<SimpleInt> siprev_t = thunkify(siprev) ;
    SimpleInt g_prev_v = g(siprev_t);
    SimpleInt g_prev_v = g2(siprev);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
  }
}
SimpleInt g(Thunk<SimpleInt> i) {
  SimpleInt icons = force(i);
  g2(icons);
```

```
}
```

**Listing 13.** Step 2: replace recursive call to g

```
SimpleInt g3(int i) {
  if (ihash <= 0) {
    return SimpleInt(42);
  } else {
    int prev = ihash - 1;
    SimpleInt siprev = SimpleInt(prev);
    g_prev_v = g2(siprev);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
  }
}

SimpleInt g2(SimpleInt i) {
  int ihash = casedefault(icons);
  return g3(ihash);
}
```

**Listing 14.** Step 3: outline block after casedefault

```
SimpleInt g3(int i) {
  if (ihash <= 0) {
    return SimpleInt(42);
  } else {
    int prev = ihash - 1;
    SimpleInt siprev = SimpleInt(prev);
    g_prev_v = g2(siprev);
    g_prev_v = g3(prev);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
  }
}
```

**Listing 15.** Step 4: replace g2 with g3

```
SimpleInt g3(int ihash) {
  int out;
  if (ihash <= 0) {
     return SimpleInt(42);
     out = 42;
  } else {
    int prev = ihash - 1;
    SimpleInt g_prev_v = g3(prev);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    SimpleInt ret = SimpleInt(rethash);
    return ret;
    out = rethash;
  }
   return SimpleInt(out);
}
```

**Listing 16.** Step 5: float out SimpleInt

```
SimpleInt g3(int ihash) {
  int out;
  if (ihash <= 0) {
    out = 42;
  } else {
    int prev = ihash - 1;
    SimpleInt g_prev_v = g3(siprev);
    int g_prev_v_hash = casedefault(g_prev_v);
    int rethash = g_prev_v_hash + 2;
    out = rethash;
  }
  return SimpleInt(out);
}
SimpleInt g2(SimpleInt i) {
  int ihash = casedefault(icons);
  return g3(ihash);
}

SimpleInt g(Thunk<SimpleInt> i) {
  SimpleInt icons = force(i);
  g2(icons);
}
```

**Listing 17.** Step 5: taking stock

# 8 Worker wrapper by local rewrites: Sum types

The transformation of floating out common patterns works equally well for branches as it does for sum types, since a case analysis on a sum type is the same as a branch on the tag.

# 9 Worker wrapper by local rewrites: Sum types + non-tail-calls

# 10 Demand analysis by rewrites: The full algorithm

# 11 Evaluation

As a baseline, we compare our performance on `nofib` test-suite, which is the test suite that GHC is tested with and performance improvements to GHC are reported on. We only consider a representative subset of programs from `nofib` ; The full test suite also extensively tests Haskell's semantics of parallelism, software transactional memory, and other features that are orthogonal to the problem of representing and optimizing non strictness.

## 11.1 A toy example: eliminating wrapper overhead

```
module {
// fact 0 = 1
// fact n = n*fact(n-1)
func @f (%i : !lz.thunk<i64>) -> i64 {
  %ival = lz.force(%i):i64
  %out = lz.caseint %ival
   [0 ->   {
    ^entry:
```

```
%x = ap(%f, %v1, ..., %vn): !lz.thunk<T>
%y = force(%x): T
```

```
%y = %f(%v1, ..., %vn): T
```

**(a)** force of a known function application: remove laziness

```
%x = thunkify(%v) : !lz.thunk<T>
%y = force(%x): T
```

```
%y = %x
```

**(b)** force of a thunk: remove laziness

```
func f(%x1, ... %txi: thunk<T>, ..., %xn) {
 %xi = force(%txi)
 %zi = ... ; %tzi = thunkify(%zi)
 f(%y1, ..., %tzi, ..., %yn)
}
```

```
func f_strict_i(%x1, ... %xi: T, ..., %xn) {
 %zi = ...
 f_strict_i(%y1, ..., %zi, ..., %yn)
}
func f(%x1, ... %txi: thunk<T>, ..., %xn) {
 %xi = force(%txi)
 %zi = ...
 f_strict_i(%y1, %y2, ..., %zi, ..., %yn)
}
```

**(c)** outlining recursive call that is immediately forced

```
func f(%x1, ... %wrapxi: @ADT, ..., %xn) {
 %xi = case (%wrapxi) [Wrapper -> ^ entry(%v) { return %v }]
}
```

```
func f_work_i(%x1, ... %xi: T, ..., %xn) {
 %zi = ...
 f_strict_i(%y1, %y2, ..., %zi, ..., %yn)
}
func f(%x1, ... %txi: thunk<T>, ..., %xn) {
 %xi = case (%wrapxi) [Wrapper -> ^ entry(%v) { return %v }]
 %zi = ...
 f_work_i(%y1, ..., %zi, ..., %yn)
}
```

**(d)** outlining of recursion of a monovariant wrapper

```
%x = constructor(@Constructor, %v1, ..., %vm)
%y = case %x
 [C1 -> {^entry1(%z11, ..., %z1n1): ... }]
 [...]
 [Ci -> {^entry1(%zi1, ..., %zini): ... }]
```

```
Inline ^entryi with %zik = %vk
```

**(e)** Case of known constructor: remove indirection

```
data C = MkC(V)
@f(%inc: C)
 case inc
 [C inv ->
 %inv = extract(@MKC, %inc) : V
 %w = ... : V; %wc = construct(@MKC, w) : C
 %rec = apEager(@f, wc)]
```

```
@frec(%inv: V)
 %inv = extract(@MKC, %inc) : V
 %w = ... : V; %wc = construct(@MKC, w) : C
 %rec = apEager(@f, wc)
@f(%inc: C)
 %inv = extract(%inc) : V
 apEager(@frec, inv)
```

**(f)** Outline recursive call of constructor that is immediately unwrapped

*BAR*

**(g)** Outline pattern matching branches on a function input

```
data C = MKC(V)
@f(...)
 ...
 %out = constructor(C, %w)
 lz.return (%out)
```

```
data C = MKC(V)
@finner(...)
 ...
 lz.return %w

@f(...)
 %w = call %f(...)
 %out = constructor(C, %w)
 lz.return (%out)
```

**(h)** Outline return of constructor

**Figure 5.** Local rewrites performed to eliminate laziness (2)

```
    %c1 = constant 1 : i64
    lz.return %c1 : i64
   }]
   [@default -> {
    ^entry:
    %c1 = constant 1 : i64
    %idec = subi %ival, %c1 : i64
    %idecthnk = lz.thunkify(%idec : i64)
    %f = constant @f : (!lz.thunk<i64>) -> i64
    %f_idec_thnk = lz.ap(%f: (!lz.thunk<i64>) -> i64,
        %idecthnk)
    %f_idec_val = lz.force(%f_idec_thnk) : i64
    %prod = muli %ival, %f_idec_val : i64
    lz.return %prod : i64
   }]
  return %out : i64
}

func @c8 () -> i64 {
  %v = std.constant 8 : i64
  return %v: i64
}

func @main() -> i64 {
  %f = constant @f : (!lz.thunk<i64>) -> i64
  %c8f = constant @c8 : () -> i64
  %c8t = lz.ap(%c8f : () -> i64)
  %outt = lz.ap(%f: (!lz.thunk<i64>) -> i64, %c8t)
  %out = lz.force(%outt) : i64
  return %out : i64
}
}
```

**Listing 18.** "Original program with wrapper and laziness overhead"

After worker/wrapper, we get the optimized program:

```
module  {
  func @c8() -> i64 {
    %c8_i64 = constant 8 : i64
    return %c8_i64 : i64
  }
  func @main() -> i64 {
    %c7_i64 = constant 7 : i64
    %f = constant @frec_force_outline : (i64) -> i64
    %c8_i64 = constant 8 : i64
    %0 = "lz.apEager"(%f, %c7_i64) : ((i64) -> i64,
     i64) -> i64
    %1 = muli %0, %c8_i64 : i64
    return %1 : i64
  }
  func @frec_force_outline(%arg0: i64) -> i64 {
    %c1_i64 = constant 1 : i64
    %f = constant @frec_force_outline : (i64) -> i64
    %0 = "lz.caseint"(%arg0) ( {
      lz.return %c1_i64 : i64
    },  {
      %1 = subi %arg0, %c1_i64 : i64
      %2 = "lz.apEager"(%f, %1) : ((i64) -> i64, i64)
      -> i64
      %3 = muli %arg0, %2 : i64
      lz.return %3 : i64
    }) {alt0 = 0 : i64, alt1 = @default} : (i64) ->
     i64
    return %0 : i64
  }
```

**Listing 19.** "factorial that has been worker/wrapper'd"

We will explain this transformation step by step. Finally, when lowered to LLVM, LLVM's strong loop analyses is able to compute a closed form for `frec_force_outline`, ending in a constant time computation.

**sid:** Explain how this happens

### 11.2 A toy example: eliminating laziness and exposing loop optimisation

```
module {
// sum up all values in the buffer
func @sum(%buffert: !lz.thunk<memref<?xi64>>) -> i64
     {
  %buffer = lz.force(%buffert) :  memref<?xi64>
  %c0 = constant 0 : index
  %N = dim %buffer, %c0 : memref<?xi64>
  %sum_0 = constant 0 : i64
  %sum = affine.for %i = 0 to %N step 1
  iter_args(%sum_iter = %sum_0) -> (i64) {
    %t = affine.load %buffer[%i] : memref<?xi64>
    %sum_next = std.addi %sum_iter, %t : i64
    affine.yield %sum_next : i64
  }
  return %sum : i64
}

// create a sequence [0..upper_bound)
func @seq(%upper_bound: i64) ->  memref<?xi64> {
  %upper_bound_ix = std.index_cast %upper_bound : i64
     to index
  %buf = alloc(%upper_bound_ix) : memref<?xi64>
  affine.for %i = 0 to %upper_bound_ix step 1 {
    %ival = std.index_cast %i : index to i64
    affine.store %ival, %buf[%i] : memref<?xi64>
  }
  return %buf : memref<?xi64>
}

func private @printInt(%i: i64)

// computes sum of numbers upto 1023?
func @main ()  -> i64 {
  %seqf = constant @seq : (i64) ->  memref<?xi64>
  %size = std.constant 1024 : i64
  %seqt = lz.ap(%seqf: (i64) -> memref<?xi64>, %size)
```

```

  %sumf = constant @sum : (!lz.thunk<memref<?xi64>>)
    -> i64
  %outt = lz.ap(%sumf : (!lz.thunk<memref<?xi64>>) ->
     i64, %seqt)
  %outv = lz.force(%outt): i64
  call @printInt(%outv) : (i64) -> ()
  %zero = constant 0 : i64
  return %zero : i64
}
}
```

**Listing 20.** "example of laziness with tensor computations"

# 12 Related Work

## 12.1 Demand analysis

[10] "Projections for demand analysis" performs demand analysis computation by modelling demands as projections on the semantic domain. We use their broad framework, adapted to our setting.

[3], [2] extend the demand analysis with finer grained information, such as Call Arity analysis.

## 12.2 GHC's intermediate representation

The Glasgow Haskell compiler [4] is a mature optimizing compiler for Haskell, whose plugin infrastructure we hook into to develop a test set for `lz` , and whose demand analysis we use as a benchmark to measure against.

## 12.3 Intel Haskell research compiler IR

The Intel labs Haskell Research compiler [8] models Core in a strict ANF language. The compiler performs demand analysis and abstract simplification on ANF. The demand analysis is performed using traditional abstract interpretation techniques. Later, this demand information is used to interpret the program and perform abstract simplification. (TODO: figure out details of this step).

It then compiles to an intermediate representation called MIL, which is a loosely typed CFG based, SSA-lite intermediate representation. They represent laziness as heap values, and manipulate the heap. Thus, their representation and usage of lazy values reasons with memory, instead of providing value semantics. MIL also does not have a notion of nested regions. Therefore, MIL extends the traditional control flow controls with more finer-grained information, called as `cut` and `interproc`.

## 12.4 Mixing dataflow analysis and transformation

This is based on the theory of compositional dataflow analysis and transformations, [6] which proves that we can interleave dataflow analyses and transformations safely.

Hoopl [9] is the dataflow analysis and transformation library within GHC. However, GHC does not use this for performing dataflow analysis over Core, as the Hoopl library is designed to work with a CFG based intermediate representation, while GHC Core is a typed functional, expression oriented intermediate representation. While Hoopl is used for certain simplifications within C– in GHC, it is not used extensively due to poor performance characteristics exhibited by the implementation. (TODO: bench Hoopl on contrived examples)

## 12.5 Alternative encodings of laziness

GRIN [1] is an alternative intermediate representation for lazy and strict functional programming languages which explicitly represents heap manipulation and case analysis. However, it is not SSA based.

## 12.6 Theoretical justification for our encoding

Our encoding is based on call-by-push-value [7], which breaks down call-by-value and call-by-name paradigms into simple primitives. In our dialect `lz` , we expose these primitives to enable a clean mixture of call by name and call by value. This enjoys the many properties that are proven to hold for call-by-push-value (TODO: which ones?!)

## 12.7 Technology substrate

MLIR [5] is a compiler framework for building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

# 13 Conclusion

We provide a baseline implementation of non-strictness for the MLIR framework which with very minimal extensions, allows us to fully express non strict semantics while being compatible with the rest of MLIR's strict dialects. As an experience report, we explore MLIR's strengths and weaknesses at representing non-strictness within an SSA based framework. We also provide a baseline implementation of demand analysis, as inspired by Wadler and Hugh's "projections for demand analysis". We evaluate our example programs against the nofib benchmark test suite. We also provide provocative examples where the potent combination of MLIR's loop optimization infrastructure along with our modest extensions allows us to beat native haskell's performance.

# 14 TODO

## 14.1 Performance benchmarking

## 14.2 Correct encoding for mutual recursion

## 14.3 Correct encoding for partially applied functions/closures

We currently have a `lz.lambda` but it's basically untested, because we didn't have many things that used closures. We need to make sure this works properly, and encodes data correctly.

**14.3.1 Lambdas in GRIN.** GRIN is also a low level IR, so it's useful to recall how this is encoded within GRIN. To quite the GRIN thesis:

" end. Using hbcc we get well optimised code in a low level functional style, comparable to for example the Core lan- guage [PJ96] used by the Glasgow Haskell compiler. The code is lambda lifted, i.e., has only super combinators, and most high level Haskell constructions, like overloading, have been transformed away. "

**14.3.2 Implementing lambda lifting.** Recall that optimal lambda lifting is $O(n^2)$. I don't know the algorithm; I'd have to study it before I implement it.

## 14.4 Stretch: Separate dialect for pattern matching

## 14.5 Stretch: rewrite based demand analysis

## 14.6 Stretch: rewrite based unification

We introduce a dialect for performing unification.

## 14.7 Stretch: rewrite based tabled typeclass unification

## 14.8 Stretch: As static as possible garbage collection

# References

[1] Urban Boquist and Thomas Johnsson. 1996. The GRIN project: A highly optimising back end for lazy functional languages. In Symposium on Implementation and Application of Functional Languages. Springer, 58–84.

[2] Joachim Breitner. 2014. Call arity. In International Symposium on Trends in Functional Programming. Springer, 34–50.

[3] Sebastian Graf. [n. d.]. Call Arity vs. Demand Analysis. ([n. d.]).

[4] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. The Glasgow Haskell compiler: a technical overview. In Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference, Vol. 93.

[5] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv preprint arXiv:2002.11054 (2020).

[6] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations. ACM SIGPLAN Notices 37, 1 (2002), 270–282.

[7] Paul Blain Levy. 2012. Call-by-push-value: A Functional/imperative Synthesis. Vol. 2. Springer Science & Business Media.

[8] Hai Liu, Neal Glew, Leaf Petersen, and Todd A Anderson. 2013. The Intel labs Haskell research compiler. In Proceedings of the 2013 ACM SIGPLAN symposium on Haskell. 105–116.

[9] Norman Ramsey, Joao Dias, and Simon Peyton Jones. 2010. Hoopl: a modular, reusable library for dataflow analysis and transformation. ACM Sigplan Notices 45, 11 (2010), 121–134.

[10] Philip Wadler and R John M Hughes. 1987. Projections for strictness analysis. In Conference on Functional Programming Languages and Computer Architecture. Springer, 385–407.

# A Appendix

Text of appendix