A static-single-assignment intermediate representation for a strict functional language

Matthew Weingarten

supervised by:
Prof. Dr. Zhendong Su
Prof. Dr. Tobias Grosser

December 2020

Abstract

Functional programming is a powerful paradigm known for its verifiability and parallelizability. When compared, however, to imperative programming languages, functional programming often lacks in performance due to fundamental memory bottlenecks. Many solutions are explored in different compiler infrastructures, but are inaccessible to each other and require high re-implementation cost. We aim to provide the groundwork for a more sustainable framework for future optimizations and to close the gap between the functional and imperative compilation processes. We present a compiler modeling a lambda calculus derived intermediate representation in a single-static-assignment based system. Furthermore, we show an implementation of a memory optimization called destructive updates using explicit reference counting techniques presented in *Counting immutable Beans*.

Contents

List of Figures

1 Introduction

Functional programming languages embody a declarative style of programming describing a computation as a chain of functions and expressions rather than a sequence of statements altering the state of the program. Each function in a purely functional language can be seen as a deterministic mathematical function and, as such, data structures in functional programming are immutable. While this has many benefits and opportunities for modularity, verifiability, and parallelizability, this can hurt the performance of functional programs. However, the main burden of performance falls onto compilers for declarative programs, as they merely describe logically what to perform and not how to perform. The difference in the compilation process manifests itself in the intermediate representation (IR) used on the path to producing machine code. While functional programs tend to use lambda-calculus-derived intermediate representations instead of the typical direct style single-static-assignment (SSA) based languages. In this thesis, we present a lambda calculus IR in a direct style SSA based framework with an extension of traditional control flow concepts allowing the representation of higher-level structures.

To further illustrate the differences between imperative and functional programming styles, we compare an implementation of a map function, squaring all elements in an array-based datastrcture in figure 1. The first map function shows an implementation in a non-specific functional language and returns a new array with the updated value, while the second map function, written in a generic imperative language, updates the mutable data located inside the array, essentially performing an update to the state of the program. While both implementations seemingly require the same amount of work, depending on the implementations and transformation to machine code, the performance may differ. With a naive approach in the functional map, creating a whole new array leads to potentially slower performance. While this would not be the case in simple examples, for larger and more complex programs the effects can become more substantial.

```
let result = map(x -> x * x,([1,2,3]))

int [] result = [1,2,3]
for(int i = 0; i < lit.size(); ++i){
    list[i] = list[i] * list[i];
}
```

Figure 1: Squaring all elements of an array in imperative and functional style

In this project, we introduce a compiler for Lean, a strict and pure functional programming language. We use a subset of *lambdapure* as the source language, an untyped functional IR used by the work in progress Lean4 compiler [1]. The current architecture of Lean on a high-level transforms a Lean program to *lambdapure* to concretize the program and lower the abstraction level. On this

level of abstraction, memory optimizations are made before lowering to a runtime environment in which the basic building blocks of Lean are implemented in C. While the optimization and the implementation details show promising results, they are potentially inaccessible to other eager and purely functional programming languages and would incur high re-implementation cost if reused in another compiling infrastructure. In this project, we want to take an attempt at modeling lambda-calculus-derived intermediate representations in the SSA-based framework, namely MLIR, such that novel optimization steps and representations of abstraction levels in future work are more accessible to other domain-specific compilers.

To show that SSA based intermediate representations are suitable to represent functional intermediate representation we present a custom *lambdapure* intermediate representation built in MLIR [2]. MLIR is a framework to create custom SSA intermediate representations, with the mission to reduce implementation costs for domain-specific IRs and provide easy transformation infrastructure and interaction with other intermediate representations. Furthermore, we perform an optimization step by inserting destructive updates, specific to the level of abstraction represented by the implemented IR. This optimization step is used by the Lean compiler and is introduced in the paper *Counting Immutable Beans* [3], from which this project takes substantial inspiration.

The goal of performing a destructive update is to reduce the number of memory allocations made during the execution of a program. It relies on the phenomena called the *resurrection hypothesis*, which is often observed in functional programs. The resurrection hypothesis describes a scenario in which an object is destroyed right before the creation of an identical or similar object. One can further observe the map function we used in figure 1, now in the Lean language.

```
inductive List
| Nil
| Cons : Nat -> List -> List

def map : (Nat -> Nat) -> List -> List
| f, Nil => Nil
| f, Cons e l => Cons (f e) (map f l)
```

Figure 2: The map function implemented in the Lean programming language

We observe that every non-last element of List is represented as a logical `Cons` object with a natural number and the list continuation associated with it. Whenever we apply a function to all elements of the List, every `Cons` object inside the list is replaced by a new `Cons` object. An application of map function on a concrete list looks as shown in figure 3.

When focusing on a single `Cons` element, we notice that the `Cons 3 Nil` object is replaced by a `Cons 9 Nil` object. The essential idea of a destructive

```
Cons 1 (Cons 2 (Cons 3 Nil)) =>

Cons 1 ( Cons 4 (Cons 9 Nil))
```

Figure 3: The map function applied to a concrete list

update is instead of allocating new memory to store the newly computed *Cons 9 Nil* object, we reuse the old *Cons 3 Nil* object and update its values. In the scenario described, we see the resurrection hypothesis being fulfilled, as the old object with value 3 gets destroyed right before the creation of the fresh object with value 9. Inserting a destructive update, in this case, would result in an execution sequence resembling an update to a mutable data structure in an imperative program and can result in better performance.

While we have shown an example of a destructive update directly in the abstraction layer of the source lean programming language, this representation is not necessarily ideal to infer when a destructive update is possible. Even more difficult would be to change the program such that a destructive update is performed, instead of a standard creation of a new object. To model this behavior, the *lambdapure* intermediate representation is much better suited. The output IR produced by the Lean compiler that is semantically equivalent to the Lean map function and looks as follows in figure 4.

```
def map (x_1 : obj) (x_2 : obj) : obj :=
  case x_2 : obj of
  L.Nil ->
    ret x_2
  L.Cons ->
    let x_3 : obj := proj[0] x_2;
    let x_4 : obj := proj[1] x_2;
    let x_5 : obj := app x_1 x_3;
    let x_6 : obj := map x_1 x_4;
    let x_7 : obj := ctor[1] x_5 x_6;
    ret x_7
```

Figure 4: Map function represented in *lambdapure* by the Lean compiler, where the *ctor[1] x_5 x_6* expression creates a new object, allocating memory

We can quickly tell that the *lambdapure* IR already contains much more information on the explicit steps required to compute the map function. Though the full details of *lambdapure* are introduced in 2.4, for this example we merely focus on the *ctor[1] x_5 x_6* statement, which calls a constructor function creating a new object, in our case, a new *Cons 9 Nil* object. At this level of abstraction, we can insert instructions to perform destructive updates, replacing the standard constructor instruction, which otherwise would not have been possible in the Lean language.

We discussed a memory optimization step on the *lambdapure* intermediate representation. More generally, declarative programming languages desire to relieve the burden of memory management from the programmer. This requires

the execution to automate assigning and freeing memory used throughout the execution of a program. Often the efficiency of the memory management system is a critical factor in the performance of a language. A common approach is to write a garbage collection system to automatically reclaim unused memory and is taken by the Haskell compiler GHC [4] or the Ocaml ocamlopt [5]. Lean on the other hand uses a reference counting system, where each Lean resource is associated with a count of how many other resources are dependent on it. In practice, however, the predominant choice in modern high-performing functional programming languages is some variation of a garbage collection technique. On the contrary, Lean uses reference counting instead of garbage collection. To keep track of references associated with each resource, the Lean compiler extends the *lambdapure* IR to insert explicit reference counting instructions, an idea adopted by the Swift IR and influencing the design choices of Lean. In our implementation, we show how an SSA infrastructure can be leveraged to insert reference counting instructions into the IR and is a great tool for these types of transformations.

Contributions. We present a Lean compiler with an SSA IR implemented in MLIR. The source language used is a subset of the *lambdapure* IR. We implement transformation passes to insert the destructive update optimization and insert explicit reference counting instructions. Our evaluations on a set of Lean programs show experimental correctness of our compiler and the performance gains of destructive updates. We conclude that a direct SSA based IR captures the high-level constructs of functional programming languages in a manner that allows for domain-specific optimizations in the same way a lambda-calculus-derived IR does.

2 Background

In this section, we introduce the background knowledge necessary for this project. We start by familiarizing ourselves with the Lean programming language and continue with describing the process of compiling functional programming languages and analyze how this process compares to the more traditional imperative style. Subsequently, we introduce the syntax and semantics of the *lambdapure* intermediate representation, the entry point of our compiler, and show lowering transformations from Lean to *lambdapure*. Finally, we introduce MLIR as the framework used to write our dialect and represent *lambdapure* in an SSA based language.

2.1 Lean

The Lean programming language is a pure and eager functional programming language and was designed with formal verification and automated theorem proving in mind. Any pure functional program describes a sequence of composing and applying a set of mathematical functions and is written in a declarative fashion. Functions are treated as first-class citizens allowing for higher-order expressions. The key defining aspect is that the output of functions must solely depend on the passed arguments and must not be affected by the state of execution. Additionally, this implies any function must not produce any side effects and all the data associated with a program is immutable. The full language and system description of Lean is found in the cited references [6, 7].

Most important to understand the compilation process and memory optimizations is the underlying type system. Lean uses an inductive type system. This means a type is defined by a list of constants and functions, such that an object can be described logically by a series of applications of its corresponding type definitions. Defining objects in this fashion allows the language to represent data-structures in an immutable and mathematical way. The natural numbers would be defined by the type definition below.

```
inductive Nat
 | 0 : Nat
 | Succ : Nat -> Nat
```

Here the number two would be created by the expression *Succ(Succ(0))*. In further type declarations, other types can be used in the declaration of more complex types. We observe a *Tree* data type to create more complex objects.

```
inductive Tree
 | Leaf : Nat -> Tree
 | Node : Nat -> Tree -> Tree -> Tree
```

Here we can represent a binary tree data-structure again in a mathematical fashion, such that `Node 5 (Leaf 3) (Node 6 (Leaf 4) (Leaf 9))` corresponds to a tree shown in figure 5. This further illustrates how data-structures in functional programming languages retain no information regarding implementation details, like defining memory layouts in structs or classes in imperative programming languages. This moves the burden of implementation to the compiler.

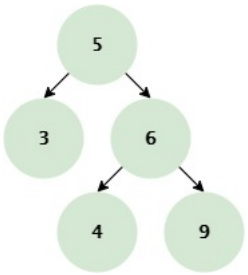

Figure 5: Tree datastructure

We further remark that it is not possible to have a cyclic data-structures in Lean. This follows from its strictness and inductive type system, such that each data type is defined by expression trees of finite height. The acyclic nature of Lean makes reference counting memory management techniques more attractive, as cyclic references are one of the main arguments against reference counting.

2.2 Comparison of lambda-calculus derived IRs to direct SSA IRs

To further understand the direct SSA based languages, we introduce the basic constructs. A program in SSA form is broken up into *basic blocks*, where each block represents a series of instructions executed sequentially without any form of control flow, meaning there are no jump statements. Each block contains a list of assignments to variables, such that each variable is assigned exactly once and assigned before it is first used. The set of Blocks are connected in a control-flow-graph (CFG) modeling the control flow of the source language. Each path in the CFG starting from the entry block corresponds to an instance of execution of the program. The blocks are connected through jump statements.

CPS-based intermediate representations on the other hand consist of a collection of basic blocks as well, called continuations. The difference is the continuations form a nested tree-like structure instead of a CFG of flattened blocks. This makes elements of the IR explicit, such as returns which are represented by a call to a continuation. In regards to functional programming, the CPS

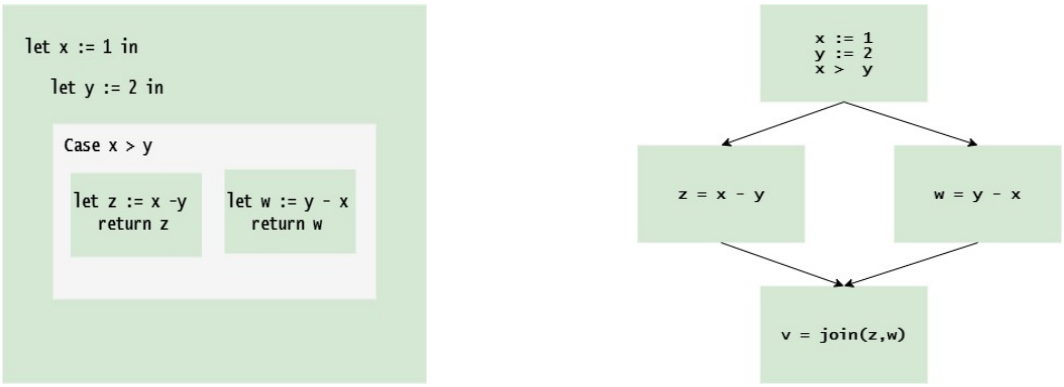

Figure 6: Simple example of the CFG in an SSA-style intermediate representation compared the continuations of the ANF form

style has clear advantages in supporting optimizations requiring higher-level knowledge, such as leveraging pattern matching cases. Eager functional languages often use the A-normal form (ANF) [8, 9] style of CPS. The A-normal form resembles a more direct-style compared to other CPS styles. *Lambdapure* IR is an A-normal form language. We further illustrate the difference focused on the structure of an ANF and SSA intermediate language.

With the addition of nested structures in a direct SSA based IR, we can capture the structure of ANF style languages with the benefits of SSA and the integration with existing architecture. The concept of *nested regions* is further introduced in relation to the MLIR framework in section 2.3. The idea of nested structures is a core design concept of the presented SSA intermediate representation and allows for transformations on the IR which would otherwise be more challenging to infer and perform.

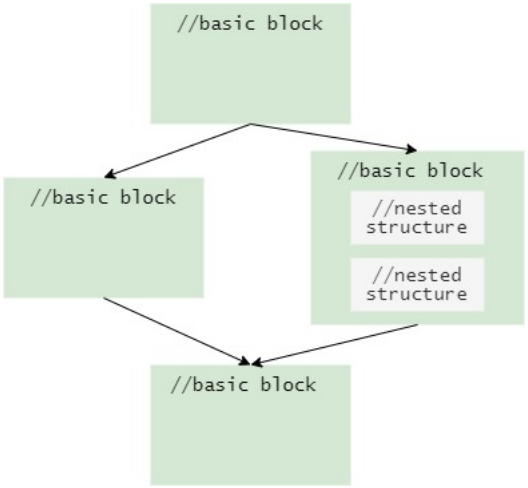

Figure 7: The nested structure extending the standard SSA control-flow-graphs allows us to combine representation of CPS-style and SSA-style representation and capture more abstract semantics

2.3 The MLIR framework

An increasing amount of language-specific compilers have called for an easier way to share implementations. Multiple different domain-specific IRs targeted LLVM-IR as a standardized intermediate step for back end code generation. However, high-level aspects of the original programming languages get lost in the lowering process and cannot be used for domain-specific optimizations. This resulted in the development of higher-level languages such as SIL IR and MIR IR used by Swift and Rust respectively. MLIR aims to reduce the burden of implementation by providing a customizable framework for SSA intermediate representations called *dialects*. The higher-level constructs, which would otherwise get lost in lower abstraction levels, such as the concept of loops, or in *lambdapure's* case, the concept of pattern matching, closures, and constructors. Not only does this standardize the process of development, but also allows for easier interaction between different IRs and progressive lowering to different layers of abstraction, each allowing for further optimizations. We introduce the basic concepts of MLIR, for more details refer to the specifications [10].

First and foremost, a *dialect* in MLIR is a representation of a custom IR and contains a set of operations, types and transformation passes on the associated dialect. Representing a program, the *module* is the top-level entity of a piece of IR code. A module holds a list of functions and their implementation. Each function has a body represented by a *region*. A region consists of a list of *blocks*, which in return holds a list of *operations*. Figure 8 shows an example MLIR program structure for an arbitrary dialect. A block forms a single unit of sequential execution in a program. The MLIR operation acts as the core element of the framework. All operations take in *values* as *operands* and each operation returns an arbitrary number of values as *results*. A value contains the results of an operation or an argument to a function or block. Additionally, all values have a type associated with them. As MLIR is a framework for SSA intermediate representations, no result can be rebound throughout the scope of a region.

```
func @name(%arg : !type) -> !type ({
 //Function region holding a list of blocks
 ^block(%block_arg : type):
  //blocks hold a list of operations
   %0 =  "dialect.operation"(%arg : !type) -> !type
   "dialect.terminator"() -> !type
})
```

Figure 8: Top level function operations represent an MLIR module, a function holds a region, a region consists of blocks and blocks contain a list of operations

Operations are the main unit of semantics in MLIR. They take and produce zero or more values. On top of that, Operations may have *attributes*. Attributes are values known at compile time and are also typed. An example operation is visualized in figure 9. The operation holds two attributes, with one being a constant integer 64 attribute with value 1, known at compile-time, and the other a symbol reference attribute, which could correspond to a function name of another function inside the module.

```
%result = "dialect.operation" (%operand1, %operand%2)
{attribute1=@symbol : type, attribute2 = 1 :i64}
```

Figure 9: An MLIR operation taking two operand values and returning a result value and has an i64 and a symbol ref attribute

Furthermore, operations may contain regions, giving us the power to represent nested regions and consequently higher-level constructs of programs in an MLIR dialect. Regions are a core concept for the *lambdapure* dialect implemented in this project. Figure 10 shows an operation holding a region, such that all operations are dominated by the holder of the region. This allows for recursive structures and higher-levels of abstraction. We use regions to model pattern matching control flow for functional programs.

```
%result = "dialect.operation" () {(

    //region held by operation

      %0 = "dialect.nestedoperation"() ({

        //nested region

        %1 = "dialect.operation"()

      })

)}
```

Figure 10: A region held by an operation allows us to create a more complex structure in our IR

To transform the IR and progressively transform towards lower-levels of abstraction, MLIR supports a pass infrastructure. This includes pattern rewrites, replacing, and inserting new operations. This lets operations describe high-level semantics and can be replaced during the lowering step with more explicit operations, as in our case done during the runtime lowering pass in section 3.6.

2.4 Lambdapure

The *lambdapure* intermediate representation is an untyped functional continuation-passing-style intermediate representation used by the Lean compiler. It is a

language in the style of A-normal form [8]. It is used as the stepping stone for further optimizations, after which *lambdapure* is translated to a C file. The following grammar [3] describes a specification of a subset of *lambdapure* used as the source language of our compiler:

$$
\begin{aligned}
&w,x,y,z \in \mathit{Var} \\
&c \in \mathit{Const} \\
&e \in \mathit{Expr} := \mathbf{app}\ \ f\ \bar{y} \mid \mathbf{pap}\ \ f\ \ \bar{y} \mid c\ y \mid \boldsymbol{ctor_i}\ \bar{y} \mid \boldsymbol{proj_i}\ x \\
&F \in \mathit{FnBody} := \mathbf{ret}\ x \mid \mathbf{let}\ x = e; F \mid \mathbf{case}\ \ x\ of\ \bar{F} \\
&f \in \mathit{Fn} := \ \lambda \bar{y}.F \\
&\sigma \in \mathit{Program} := \mathit{Const} \to \mathit{Fn} \\
&(1)
\end{aligned}
$$

We remark that $\bar{y}$ is a variable set of expressions, meaning app $f\ \bar{y}$ is a function application on an set of variables $\bar{y}$ of arbitrary length.

Figure 11: Lambdapure specification

A *lambdapure* program consists of a list of function implementations bound to their constant name. Each function consists of statements, either a return, assignment, or a case statement chained together. Case statements are the only form of control flow in *lambdapure* and all leaves of control flow branches terminate in a return statement. In lambdapure, all data types are abstracted to a single object type. An object in *lambdapure* can represent any data object or a closure, allowing for higher-order functions. A `case` statement on an object evaluates to the i-th branch corresponding the $\texttt{ctor}_i$ expression used in the creation of this object. An object can be created by a $\texttt{ctor}_i$ expression, storing arguments as fields. The i-th field is accessed using the projection expression $\texttt{proj}_i$. The expression `app f y` evaluates by applying function f to y, requiring a full application. In contrast, `pap f y` evaluates to another lambda expression, therefore a closure. As intermediate representation *lambdapure* provides the following guarantees: The language is already type-checked, all constructor applications are fully applied, and some optimizations are already performed, like dead code elimination. Additionally, all function abstractions are lifted to top-level constants.

Figure 12 shows the translation of a Lean function swapping the left and right subtrees of a tree node to *lambdapure*. While some type of information such as `Tree.Nil` and `Tree.Node` on line 1 and 5 respectively are kept, it is not essential information in compiling *lambdapure* and can be replaced by the respective object tags. It is important to note that the type declaration when calling a constructor is no longer used on the level of *lambdapure* IR. Only a

tag is stored inside the object to denote which constructor of a type is used, so whenever a node of type tree defined in figure 12 is created the tag is set to one, while for nil of type tree the tag is set to zero, implying the type of an object (whether an object is of type tree or type list) is no longer usable for transformation at this layer of abstraction. This has implications on the implementation of detecting candidates for destructive updates in section 3.4.

```
inductive Tree
 | Nil
 | Node (l r : Tree) : Tree


def swap : Tree -> Tree
 | Nil =>  Nil
 | Node l r => Node r l


def swap (x_1 : obj) : obj :=
  case x_1 : obj of
  Tree.Nil ->
   ret x_1
  Tree.Node ->
   let x_2 : obj := proj[0] x_1;
   let x_3 : obj := proj[1] x_1;
   let x_4 : obj := ctor_1[Tree.Node] x_3 x_2;
   ret x_4
```

Figure 12: Transformation from a swap function in Lean to a representation in *lambdapure*

3 Representing lambdapure in SSA

In this section, we introduce the details of the *lambdapure* dialect. The *lambdapure* dialect captures the semantics of the lambdapure intermediate representation described in 2.4 of the Lean compiler and is extended to perform destructive updates, reference counting, and lowering to a runtime representation. Additionally, the MLIR type system is extended with a *lambdapure* specific type, namely an object type visualized in figure 14. In figure 13 we show the complete architecture of the *lambdapure* dialect including all operations and passes. We continue by introducing the type system, then the standard operations, and consecutively the three passes and their associated operations. The implementation is found on GitHub [11]. The code listings shown are modified for readability and do not correspond to the explicit syntax used in the implementation, but convey the full semantics. Also, all operations and functions have associated type information, which we omit as almost all types are of object type and would reduce the clarity of examples if shown.

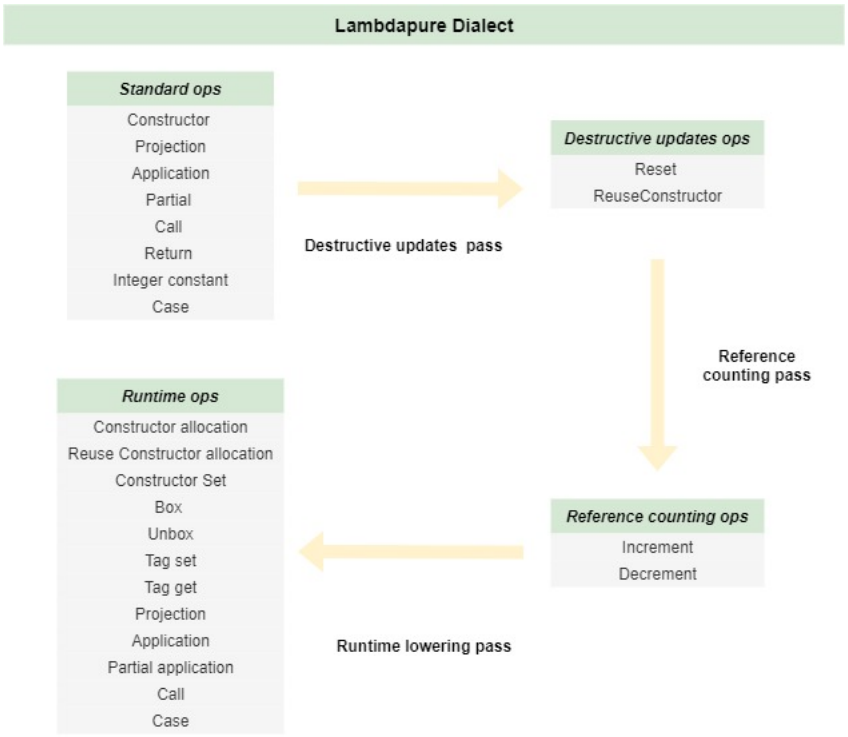

Figure 13: Lambdapure dialect architecture shows the set of all operations, the three transformation passes, and how they are organized

3.1 Type system

We introduce a new type system to represent a *lambdapure* program. With the *lambdapure* intermediate representation this is straightforward as any value is boxed in a single type; the object type. This means we add a custom *lambdapure* `ObjectType` to our dialect. The *lambdapure* object wraps closures, constructor objects for custom Lean data types, and the built-in data type *Nat* for natural numbers, which can be seen as a special form of a Lean data type. Further, we use the MLIR built-in `IntegerTypes` to describe primitive integer types used for internal implementations like object tags and constructors.

```
%x = "op"(%arg0, %arg1) {attribute = 1 : i64} : (!Object, !Object) -> !Object
```

We observe an operation example with fully expanded type information taking in operands of object type and resulting in a value also of object type.

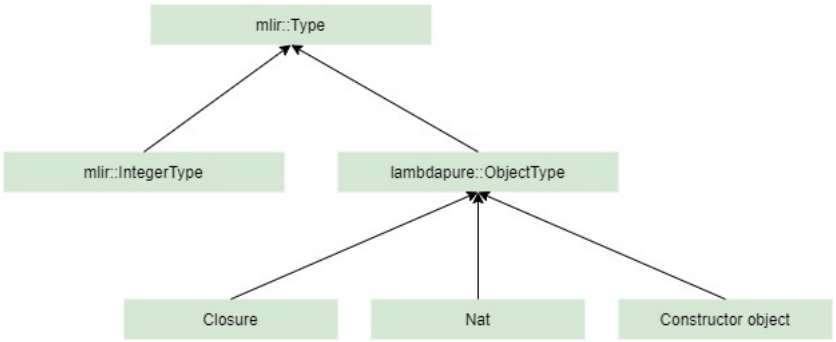

Figure 14: Lambdapure dialect type system

3.2 Standard Operations

To describe the core functionality of a *lambdapure* and therefore Lean program we create the set of standard operations. The standard operations have a one-to-one relationship with the *lambdapure* intermediate representation introduced in the *lambdapure* specification in figure 11. The *lambdapure* dialect in this form resembles an SSA-based language, however does not introduce any control flow in the form of jumps common in the direct style intermediate representation, instead uses MLIR regions to represent different branches of execution. The set of standard operations describe the semantics of any *lambdapure* program and is used as our starting point for our destructive updates, reference counting, and runtime lowering pass.

3.2.1 Constructor operation

The constructor operation represents the creation of a Lean data type, resulting in a *lambdapure* object. Here, the type information is stored in the tag attribute required for pattern matching in case operations. It takes a variable amount of

operands of type objects as data to be held within the object. The following listing shows an example constructor operation taking as inputs two object values *%arg0* and *%arg1* and storing a tag of value one alongside them. The resulting object created by the constructor operation is what we refer to as a constructor object, meaning this object holds a piece of a *lambdapure* data defined by its type. This can correspond to creating a node in a binary tree of a lean data type $Node: \rightarrow Tree \rightarrow Tree \rightarrow Tree$. Important to note is that all objects share a single constructor operation irrespective of the Lean data type created, with the tag being the only distinguishing factor between constructors. The constructor with tag attribute operation corresponds to a $ctor_i$ expression in the *lambdapure* intermediate representation.

```
%x = ConstructorOp( %arg0,%arg1 ) [tag = 1]
```

3.2.2 Projection operation

The projection operation represents access to arguments of an object resulting from a constructor operation. The object to be accessed is determined by the operand of the projection and the argument is determined by the attribute index. The example shows access to the second argument of the constructor operation that created the variable *%x*. This example could correspond to access the right subtree of a Node object. Together the projection and the constructor operation describe the core operations to represent any Lean data structure.

```
%x = ProjectionOp(%arg0) [index = 1]
```

3.2.3 Case operation

The case operation is the only form of control flow present in the *lambdapure* Dialect. The input operand is an object on which a pattern match is performed. The operation holds a list of regions, each of which corresponds to the implementation of a pattern-matching branch. Given an input operand of an object created by a constructor operation with tag i, the case operation evaluates the i-th region. Each branch in the case operation forms an independent unit of operations. The function *equalsZero* illustrates how a case statement is used to implement a function *if x == 0 then true else false*.

```
func @equalsZero(%x)
%0 = IntegerConstOp() [value = 0]
%1 = callOp (%x,%y)   [@callee = @Nat.equals]
CaseOp(%1) {
    %1 = ConstructorOp () [tag = 0]
    ReturnOp (%1)
}{
    %2 = ConstructorOp() [tag = 1]
    ReturnOp (%2)
}
```

3.2.4 Application operation

The application operation models higher-order functions in our dialect. The first operand in an application operation is always a closure object, while the rest of the operands represent arbitrary constructor objects. The constructor objects get applied to the closure. Since the application operation requires a full application, the resulting object is also a constructor operation. Ultimately this allows lambdapure to treat functions as first-order citizens which can be assigned to a value in our dialect. A higher-order function like *let applyTwice foo x := foo(foo(x))* is represented in our dialect as shown.

```
func @applyTwice(%foo, %x){
    %0 = AppOp (%foo, %x)
    %1 = AppOp(%foo,%0)
    ReturnOp(%1)
}
```

3.2.5 Partial application operation

Similar to the application operation, the partial application operation also represents a higher-order function construct, but in contrast, represents a function application that returns a closure. As the name suggests any full application is represented by application operation, and any partial application by a partial operation. An example of a closure creation by a partial application operation is visualized below and represents a closure produced by *addOne x := x + 1*.

```
%x = IntegerConst() [value = 1]
%addone = PapOp(%x ) [callee = @add]
```

3.2.6 Further operations

The last three operations, the **call operation**, the **return operation** and the **Integer constant operation** are simple operations required to fully represent a *lambdapure* program, where the call operation describes a standard call to a function determined by the *callee* attribute, passing the operands as arguments. The integer constant operation and return operation represent a statically known *Nat* value and the end of a function execution respectively.

```
%x = IntegerConst() [value = 1]

ReturnOp(%x)

%x = CallOp(%arg0, %arg1) [callee = @map]
```

3.3 Passes

Overall we perform three consecutive passes for full lowering of the initial module generated by the *lambdapure* IR. First, we apply transformations to insert reset and reuse operations in the destructive update pass, the main optimization step. Next, we insert explicit increment and decrement operations for reference counting of the objects. This has to be performed strictly after the destructive updates pass as the insertion of reuse and reset operations affects the insertion of the reference counting operations. Lastly, we perform a runtime lowering pass to create an MLIR module using runtime implementations of the Lean runtime library in preparation for translation. All passes introduce new operations.

3.4 Destructive update pass

The goal of the destructive updates pass is a reduction of the amount of memory allocated in our *lambdapure* program. Performing destructive updates is based on the resurrection hypothesis, namely, whenever an object dies right before the creation of a similar object. Initially, the pass statically infers candidate constructor objects that are suitable for destructive updates on preexisting objects. Even still, both the unshared and the shared branch of execution must be accommodated, since the behavior varies depending on the result of the runtime reference counting check. To model this behavior we introduce two new operations, the reset and reuse constructor operation, to be inserted during the destructive updates pass.

3.4.1 Reset and reuse constructor operation

The reset operation takes in an object as an operand that is a candidate for reuse in a future constructor operation. The reset operation introduces a new control flow operation, as it contains two regions. The reset operations represent a runtime check of the reference count determining whether the reset object is in a shared or unshared state. The first region describes the series of operations executed in the scenario where the reuse object is shared and the second region dominated by the reset operation describes the standard program behavior for the shared state of the reset object. The transformation in figure 15 shows an example of a destructive update insertion.

The reuse constructor operation is designed to replace the standard constructor operation in case a candidate for memory reuse is found. A valid reuse constructor operation must always be nested inside the first region held by a reset operation. It adds the candidate reset object as an operand to the operation, which is repopulated with the new data values and tag, effectively destroying the old object and replacing it with the new one. This operation performs the creation of an object without ever allocating new memory, whereas the standard constructor operation will always allocate a new chunk

of memory for the fresh object. This operation essentially performs the destructive update, effectively destroying the old object set for reuse. Together these two operations allow the resulting program to handle both the shared and unshared runtime state and the reuse incurs zero cost when compared to a stand constructor.

3.4.2 Inserting reset and reuse

The destructive update pass inserts reset and reuse operations when constructor operation has been declared to be suitable for a destructive update. To insert a destructive update we must first find a candidate object that can be reused and pair it with a constructor operation that reuses. The candidate and the matching constructor operation have to fulfill the following criteria to be deemed a valid destructive update combination: (1) The candidate may never be used by any operation of all execution branches succeeding the paired constructor operation, and (2) the size of the reuse object must be smaller or the same size as the result of the constructor operation. Criteria (1) guarantees that no other operation requires the values of the object after its destruction, which would result in an incorrect computation, where the operation expects the old values in the reset object, but uses the new ones instead.

```
func @example(%arg0)  {
    %x_1 = ConstructorOp(%x_2,%x_3) [tag = 1]
    ReturnOp(%x_1)
}

func @example(%arg0) {
  ResetOp(%arg0) {
    %x_1 = ConstructorOp(%x_2,%x_3) [tag = 1]
    ReturnOp(%x_1)
  },  {
    %x_1 = ReuseOp(%arg0,%x_2,%x_3) [tag = 1]
    ReturnOp(%x_1)
  }
}
                        .
```

Figure 15: Inserting a reset and reuse operation around the standard constructor operation. Notice that the *ReuseOp* uses memory *%arg0*, which is the same memory captured by the *ResetOp*

The pass is applied to each function in the module. It begins by initializing a list of potential reuse candidates with the arguments of the function before traversing the function region by region, starting with the region owned by the function operation. When a case operation is encountered, each region of the case operation is traversed independently, with a copy of the current candidate list propagating to all subregions. Whenever a constructor operation is encountered, an opportunity for reuse constructor insertion is explored. Our candidate list is scanned for potential matches fulfilling criteria (1) and (2). Criteria (1) is checked using additional candidate size inference described in section 3.4.3 and criteria (2) is checked by scanning all subsequent operations,

including nested regions dominated by the current region. If the criteria are met, the parent region of the found constructor op is copied twice into the region owned by a newly created reset operation with one copy replacing the standard constructor operation for a reuse constructor operation. For successive regions, the candidate used in the reset and reuse operations is removed from the candidate list. The transformation in figure 15 demonstrates an application of the destructive updates pass.

3.4.3 Reuse candidate inference

On the level of abstraction of our *lambdapure* dialect, we are oblivious to the size of an object passed to a function, making it difficult to infer whether the size of the reset object is smaller than the size of our fresh object, a requirement that must be fulfilled for correct insertion. While this can potentially be done using dynamic size checking of the object, we want to circumvent the overhead by statically deciding the lower bounds of sizes for the reuse candidates for each region. Therefore to ensure the capability for correct reuse of an object we must infer the size based on the operations present inside the region of potential reuse. Additionally, we do not know if the object passed represents a closure, which would be unsuitable for reuse.

To guarantee the correctness of a destructive update we initially treat all function inputs as potential candidates and start the analysis by walking over all operations in a function, including all the operations nested within subregions. If an object in the candidate list is not used by a projection operation at any point in the function, we can discard the object from the candidate list. The reasons for this are twofold: First, if the object is used as an operand in the projection operation we guarantee the object cannot be a closure, and second, there is information in the function to extract the size of an object statically, which without the projection operation would not be possible. Subsequently, we perform a per region pass with the filtered candidate list attempting to resolve the size of the object, which evaluates to the maximum index accessed by a projection operation. The information gathered from this pass and the candidate list at the entry point is fed to the destructive update pass.

```
func @map(%f,%cons) {
  CaseOp(%cons)  {
    ReturnOp(%cons)
  },  {
    %1 = ProjectionOp(%cons) [index = 0]
    %2 = ProjectionOp(%cons) [index = 1]
    %3 = AppOp(%f, %1)
    %4 = CallOp(%f, %2) [callee = @map]
    %5 = ConstructorOp(%3, %4) [tag = 1]
    ReturnOp(%5)
  }
}
```

Figure 16: The only reuse candidate in the map function is *%cons*, since *%f* is never used by a projection

We use the map function in figure 16 to illustrate the candidate inference technique. Initially, our candidate list holds the argument objects *%f* and *%cons*. After finding no projection operations using *%f*, we remove *%f* from the candidate list. The candidate inference pass correctly determines that *%f* is not suitable for reuse, and rightfully so, as *%f* represents a closure. After projection filtering, the remainder of our candidates, in our case only *%cons*, are guaranteed to hold constructor objects and cannot hold closures. Closure objects are unsuitable for memory reuse for their in-memory representation differs from that of a constructor object.

Once our candidate list only consists of constructor objects, a size is associated with each candidate object on a per-region basis to determine whether our fresh object fits into the allocated memory slot of the original object. The example in figure 16 has three regions: (1) the region representing the body of map, the first (2) and the second (3) case regions. In regions (1) and (2) no projection operations are present, defaulting the size of *%cons* to zero. Assume a constructor operation with one or more operands was present in either of the two regions, a destructive update could not be inserted as the size of the original object might not be large enough to hold the data of our fresh object. On the contrary, in region (3) we find a projection operation with an index of one, confirming the original object *%cons* has a size of at least two. The result of the destructive update pass in region (3) is shown in figure 17.

```
func @map(%f, %cons){
  CaseOp(%cons)  {
    ReturnOp(%cons)
  },  {
   ResetOp(%cons)  {
     %1 = ProjectionOp(%cons) [index = 0]
     %2 = ProjectionOp(%cons) [index = 1]
     %3 = AppOp(%f, %1)
     %4 = CallOp(%f, %2) [callee = @map]
     %5 = ConstructorOp(%3, %4) [tag = 1]
     ReturnOp(%5)
    },{
     %1 = ProjectionOp(%cons) [index = 0]
     %2 = ProjectionOp(%cons) [index = 1]
     %3 = AppOp(%f, %1)
     %4 = CallOp(%f, %2) [callee = @map]
     %5 = ReuseOp(%1,%3, %4) [tag = 1]
     ReturnOp(%5)
   }
  }
}
```

Figure 17: The resulting map function after performing a destructive updates pass and inserting reset and reuse on the object *%cons*

3.4.4 Limitations

The algorithm described finds merely a subset of all possible insertions of destructive updates. In situations where projections are not performed before we construct a new object, we miss out on optimization opportunities. Take

the example of a Lean program in figure 18 where the right subtree of a node is replaced by the left subtree. Theoretically, a destructive update could be performed, but it does not get detected in our destructive updates pass. This suggests higher-level representations of the Lean program are beneficial for future improvements, especially concerning additional type information, before lowering everything to be modeled as an object. The type information could be leveraged to allow size inference without relying on projections.

```
def copy : Tree -> Tree
    | Nil =>  Nil
    | Node l r => Node l l

func @copy(%node) {
  CaseOp(%node)  {
    ReturnOp(%node)
  },  {
    %1 = ProjectionOp(%node) [index = 0]
    %2 = ConstructorOp(%node, %node) [tag = 1]
    ReturnOp(%2)
  }
}
```

Figure 18: As the right subtree is never accessed we are unable to infer whether the *%node* is a reusable object with enough space

We remark that in cases where destructive objects are performed and the fresh object is strictly smaller than the original object, memory fragmentation occurs. Allowing for insertion in these cases results in a trade-off choosing to forego a destructive update and avoid fragmentation or allocate a new chunk of memory, incurring higher runtime but keeping objects in memory contiguous. An example is shown in figure 19. Here a destructive update would be performed on the pattern matching case of *Node3*, the fresh object would be smaller than the original leading to wasted memory in the allocated slot for the original object for a *Node3* type, causing memory fragmentation. Further evaluation is discussed in section 4.

```
inductive Tree
 | Nil
 | Node2 (l r : Tree) : Tree
 | Node3 (l m r : Tree) : Tree

def cut : Tree -> Tree
 | Nil => Nil
 | Node2 l r => Node2 (cut l)  (cut r)
 | Node3 l m r => Node2 (cut l)  (cut m)
```

Figure 19: Memory fragmentation occurs if destructive updates are performed in the pattern matching case of the *Node3* object. The *Node2* object is smaller in size than the *Node3* object, thereby leaving unused chunks of memory on replacing *Node3*'s content with *Node2*

3.4.5 Tag setting of reused objects

The last step to be addressed for inserting destructive updates is resetting the tag of the reused object. Often the original tag and the new tag match, implying updating the tag is not necessary. This is not always the case, as seen in the cut function in figure 19 where a *Node3* object is replaced by a *Node2* object, requiring a tag set. A scenario with a matching tag is always materialized whenever the constructor operation marked for reuse resides in the i-th arm of a case statement on the reused object. Concretely we insert a tag set operation for any reuse constructor that is not located in a region belonging to a case operation, or the value of the tag attribute does not match the case region number of the closest case operation on the reused object. Overall this saves the small overhead incurred by setting the tag to the same previous value.

3.5 Reference counting pass

To perform memory management throughout the duration of a program Lean associates each object with a reference count to be updated throughout execution. Additionally, the information gained by keeping track of references to an object allows us to perform exclusivity checks during runtime, essential in guaranteeing correctness when applying destructive updates. The reference counting pass transforms our program to correctly keep track of the reference counts by inserting explicit reference counting operations.

The reference counting mechanism in Lean provides an implicit contract for the effect of different operations on the reference counts of their arguments. We provide a table listing the effects and discuss the ramification thereof on our IR design and insertion of explicit reference counting operations. The effects of the reset and reuse operations are discussed in more depth in section 3.5.1.

- Call/Application/Partial application: increases the reference count of all arguments
- Return: decreases the reference count of the returned object
- Constructor: increases the reference count of each of the objects passed to the constructor operation
- Case/Projection : No effect on the reference count
- Reset: decreases the reference count of every object held by the reset object
- Reuse constructor: increases the reference count of each of the objects passed to the reuse constructor, except for the first operand which signifies the object to be reused. The value for the reused object and the result value of the reuse operation are treated as a single entity in further analysis, as they refer to the same memory

Furthermore, objects passed as arguments to a function experience a different behavior. In Lean, a function takes *ownership* of the single reference count increase required to call the function/closure, which follows from the effect of the call, application, and partial application operations. We also refer to an entity of reference count as a *token*. More concretely, ownership means the function is responsible for consuming the tokens of all arguments once the execution of the function has ended.

To model the reference counting mechanisms correctly we introduce two new operations to be inserted during the reference counting pass. The **increment** and **decrement** operation explicitly increment and decrement the reference count of an object passed as an operand respectively. We illustrate this with a few examples:

In this case of the *id* function, no reference counting operations must be inserted as the function just consumes the reference exactly once when returning the passed object.

```
func @id(%x) {
  ReturnOp(%x)
}
```

In contrast to the *id* function, the *get* function must decrement the reference count of object *%x* before returning. Our pass determines that no operation inside *get* consumes the token of *%x* and as the function is responsible for consuming a token, a decrement operation must be inserted. If no insertions were made, calling the get function would permanently increase the reference count, even after the function has terminated and no longer points to the object.

```
func @get(%x) {
  %y = ProjectionOp(%x) [index = 0]
  DecOp(%x)
  ReturnOp(%y)
}
```

To motivate the need for the increment operation, the *create* function shows an example where the *%x* object is consumed twice. The constructor operation consumes a token for each operand, in this case, it consumes the token for *%x* twice, with both fields of the constructor object pointing to *%x*. This requires insertion of an increment operation as the *create* function has only one reference token to spend, due to ownership, and must acquire a further token with an increment operation before the constructor operation.

```
func @create(%x) {
  IncOp(%x)
  %y = ConstructorOp(%x,%x) [tag = 0]
  ReturnOp(%y)
}
```

The rules described above must hold for every possible branch of execution. This means the reference counting pass is run on a region-by-region basis. Concretely, when a case operation is encountered during our pass, the current consumption information of all objects is propagated to the regions owned by the case operation. This can result in different operation insertions in regions, as seen in a slightly more complicated example in figure 20, where an object is updated by applying a closure to the value held by an object only in the second pattern matching case.

```
func @update(%f,%x) {
  CaseOp(%x){
      DecOp(%f)
      ReturnOp(%x)
  }{
      %y = ProjectionOp(%x) [index = 0]
      IncOp(%y)
      %z = AppOp (%f, %y)
      %w = ConstructorOp(%z) [ tag = 1]
      DecOp(%x)
      ReturnOp(%w)
  }
}
```

Figure 20: The first region of the case operation requires an insertion of a *DecOp(%x)*, while the second region requires an insertion of an *IncOp(%y)* and *DecOp(%x)*

The first region of execution merely decrements *%f* as no operations in the region consume *%f* before returning. On the other hand, *%x* is an operand to the return operation. This implies the reference is still live outside the scope of the function and the return operation consumes the token of *%x* inside the *update* function. In the second region, we must insert an increment operation for value *%y* before passing it as an operand to the application operation. We observe both *%f* and *%y* get passed to the operation, but only *%y* is incremented. The difference is, *%f* is a function argument and already has a token associated with it, while *%y* is the resulting value of an access to an object held by the constructor object *%x*. As a direct consequence, the value *%y* has no tokens before passed to the application operation and requires an increment. Additionally, a decrement operation on *%x* has to be inserted as no operations consume a token before termination of *update*.

To summarize, the reference counting pass traverses every possible execution path of the program. During the walk over all operations, the pass keeps track of the consumption on each value defined by the rules introduced above and inserts decrement and increment operations whenever necessary. This results in the minimal possible reference counting operations inserted. While a decrement and an increment on the same value would cancel each other out and could be removed, this never is manifested in the IR as the pass keeps track of consumption on each value before insertion.

3.5.1 Reference counting with destructive updates

After performing a destructive update pass the original region gets split into two regions dominated by a newly created reset operation. To even further motivate the necessity of the reset operation and the design choice of using two almost identical regions, we illustrate the difference in reference counting instructions when comparing the reset and non-reset region. The first subregion of the reset operation describes the sequence of operations executed if the reset object is determined to be non-shared at runtime. Consequently, the first subregion effectively destroys the reset object, implying the references held by the reset object are removed, and the reference counter thereof must be decremented. In regards to the reuse constructor operation, it gets handled almost the same as the constructor operation, where all operands consume a token. However, the first operand describes the reset object. Instead of counting the first operand of the reuse constructor as consumption, we treat the resulting value and the first operand as a single entity for the rest of the transformation pass. Intuitively this makes sense, as the result of the reuse constructor points to the reuse object.

```
 func @swap(%node){
  ResetOp(%node)  {
    %0 = ProjectionOp(%node) [index = 0]
    %1 = ProjectionOp(%node) [index = 1]
    %2 = ConstructorOp(%1, %0) [tag = 1]
    ReturnOp(%2)
  },  {
    %0 = ProjectionOp(%node) [index = 0]
    %1 = ProjectionOp(%node) [index = 1]
    %2 = ReuseOp(%node, %1, %0) [tag = 1]
    ReturnOp(%2)
  }
}

 func @swap(%node){
  ResetOp(%node)  {
    %0 = ProjectionOp(%node) [index = 0]
    %1 = ProjectionOp(%node) [index = 1]
    %2 = ReuseOp(%node, %1, %0) [tag = 1]
    ReturnOp(%2)

  },  {
    %0 = ProjectionOp(%node) [index = 0]
    %1 = ProjectionOp(%node) [index = 1]
    IncOp(%0)
    IncOp(%1)
    %2 = ConstructorOp(%1, %0) [tag = 1]
    DecOp(%node)
    ReturnOp(%2)
  }
}
```

Figure 21: Effect of reset and reuse operation on reference count insertions

We show an example of a reference counting pass applied to a program in which a destructive update was inserted in the transformation in figure 21, corresponding to the swap function swapping the left and right subtrees of a

node. After the destructive updates pass, the reset operation has two subregions, one for the destructive update and one for the standard update. The second subregion is as expected and follows the standard reference counting insertion rules introduced before the special case of destructive updates. In contrast, the first region requires no explicit reference counting instructions. The reset operation decrements the reference of objects held by the reset object *%node*, canceling out the increment incurred by passing it to the reuse constructor operation. Also, the decrement operation is no longer required, since we are returning and consuming a reference token of the original *%node* object indirectly by returning the result of the reuse constructor operation.

To further convince that the fusion of the reset and reuse operation would not work, we show an example using only a reuse constructor operation without the reset operation. Assume the reuse constructor operation checks for exclusivity right before constructing an object instead of at the beginning of the region in which it was inserted. This would result in the program in figure 22. Adding increment and decrement operations would lead to incorrect reference counts in the case *%node* is not shared and destructive update is performed. The *%node* object would deflate its reference count by one and the objects held by node would have inflated their reference count by one incorrectly. Similar issues occur in the case where we decide to not insert the increments and the decrements in the case where the *%node* is not shared.

```
func @swap(%node){
   %0 = ProjectionOp(%node) [index = 0]
   %1 = ProjectionOp(%node) [index = 1]
   IncOp(%0)
   IncOp(%1)
   %2 = ReuseOp(%node, %1, %0) [tag = 1]
   DecOp(%node)
   ReturnOp(%2)
}
```

Figure 22: The swap function without a reset operation would result in faulty reference counting in either the shared state or the unshared state and can never handle both cases

Succinctly, the reference count insertion pass has to be performed strictly after the destructive updates pass and ensures the reference counts associated with each object reflect the correct state of references throughout the program execution. The reference counts act not only as an essential element of memory deallocation but also work hand in hand with the destructive update pass to guarantee the correct application of the destructive updates optimization.

3.6 Runtime lowering pass and translation

The Lean runtime library [12] implements the built-in environment and a set of routines invoked by the output C file modeling the fundamental behavior in Lean and therefore *lambdapure*. The Lean runtime and the result of our MLIR compiler together can be compiled to produce executable code. The runtime pass lowers our dialect to conform to the runtime environment. In the runtime environment, any object is represented as a struct Lean object, a wrapper struct for all object types. For the scope of this paper, we limit lean objects to represent constructor objects and closures. The low-level routines for this implementation are:

`alloc_ctor`: allocating memory for a Lean object given size and tag

`ctor_get` and `ctor_set`: getting and setting the i-th index of a constructor object

`ctor_tag_get` and `ctor_tag_set`: getting and setting the tag of a constructor object

`alloc_closure`: creating a closure object

`apply_i`: applying a closure object with i arguments

`inc_ref and dec_ref`: incrementing and decrementing the reference of a Lean object

`is_exclusive`: checking exclusivity of a Lean object

`box` and `unbox`: boxing and unboxing primitive integer types to wrap them as Nat, a subtype of Lean object

`nat_add` and more: built-in standard functions for Nat, such as addition, multiplication, and comparison

During the runtime lowering pass, we insert instructions to make the program more explicit, and the resulting module resembles the targeted C file. Representing the runtime code as a *lambdapure* module instead of translating the higher-level of abstraction of the standard operations simplifies the process of bug-fixing and also of translation, such that every operation can be translated independently. Also, any future updates to the runtime environment that do not fundamentally change the behavior require potential reimplementation only of the affected operations.

While some operations already correspond to an invocation of a runtime function, we introduce new operations producing a more explicit intermediate representation. The allocation and the reuse allocation operation to allocate memory or reuse memory respectively for the object created by the constructor. The constructor set operation inserts arguments of the constructor operation

into the allocated slot. Additionally, we add an object tag setter and getter operation and boxing and unboxing of primitive types to the built-in data type Nat.

```
def filter : List -> List
  | Nil => Nil
  | Cons n l => if n > 5 then filter l else Cons n (filter l)

func @filter(%list) {
    case (%list){
        returnOp (%list)
    },{
        %1 = ProjectionOp(%list) [index = 0]
        %2 = ProjectionOp(%list) [index = 1]
        %3 = IntegerConstOp()    [value = 5]
        %4 = CallOp(%3,%1)       [callee = @Nat.decLt]
        CaseOp(%4) {
            %5 = CallOp (%2)     [callee = @filter]
            %6 = ConstructorOp (%1,%5) [tag = 1]
            ReturnOp(%6)
        },{
            %7 = CallOp(%2)     [callee = @filter]
            returnOp(%7)
        }
    }
}
```

Figure 23: The filter function removing all elements from a list smaller than five, shown before runtime lowering

In figure 23 we illustrate an application of the runtime lowering pass to best describe the behavior of the pass. The filter function removes all elements in a list smaller than five. For sake of understanding the runtime lowering, we omit the destructive and reference counting pass. However, this does not change the primary insertions and produces a valid runtime file no matter the passes applied.

Tag getter operations are inserted as an assistant to case operations on the runtime level. In figure 24 a tag getter operation is inserted before the case statement. The second case operation on value *%4* on the other hand does not require an object tag getter as the call to *Nat.decLt* returns a primitive integer type instead of an object. In summary, after a runtime lowering every case operation must take an integer type as an operand instead of an object.

The constructor operation is replaced by a memory allocation operation with a predefined size and tag attribute. Essentially the standard constructor operation is unrolled into allocation and field setting, done by the constructor set operation. The purpose of this at this time is solely to conform to the runtime library and making the translation easier. The same applies to the box operation.

```
func @filter(%list) {
    %0 = TagGetOp(%list)
    case (%0){
        returnOp (%list)
    },{
        %1 = ProjectionOp(%list) [index = 0]
        %2 = ProjectionOp(%list) [index = 1]
        %3 = BoxOp()             [value = 5]
        %4 = CallOp(%3,%1)       [callee = @Nat.decLt]
        CaseOp(%4) {
            %5 = CallOp (%2)     [callee = @filter]
            %6 = AllocCtorOp()   [size = 2, tag = 1]
            CtorSetOp(%6,%1)     [index = 0]
            CtorSetOp(%6,%5)     [index = 1]
            ReturnOp(%6)
        },{

            %7 = CallOp(%2)     [callee = @filter]
            returnOp(%7)
        }
    }
}
```

Figure 24: The filter function after applying the runtime lowering pass strongly resembles the runtime environment with the addition of the *TagGetOp*, *AllocCtorOp* and *CtorSetOp* operations

Finally, the last step of lowering to the targeted C file is the translation of the produced *lambdapure* module after the runtime pass. As stated previously, translation is straightforward as the other passes do the heavy-lifting, and all operations correspond to a single call or statement in the runtime library: figure 25 shows a few translations. The callee attribute of the call operation is first checked for a direct call to a runtime invocation. Observe the call operation with attribute `@Nat.declt`, which gets translated to a built-in lean function called `lean_nat_decLt`, while the callee attribute of `@filter` translates to a function call defined in the program. To get a better picture of the passes, figure 26 shows the iteration of the map function throughout all pass infrastructures and the C file after lowering in full detail.

```
%x = AppOp(%arg0,%arg1)                       => obj* x = apply_1(arg0,arg1)
%x = AllocCtorOp() [size = 1, tag = 1]        => obj* x = alloc_ctor(1,2,0)
%x = CallOp(%arg0,%arg1)[callee = @Nat.decLt]=> int  x = lean_nat_decLt(arg0,arg1)
%x = CallOp(%arg0) [callee = @filter]         => obj* x = filter(arg0)
```

Figure 25: A few example translations showing the correspondence of runtime invocations to operations after the runtime lowering pass

3.7 Parser and MLIR generation

Lambdapure is a language with a context-free LL grammar defined in figure 11. To parse any *lambdapure* file we use a Top-down LL(1) parser, handling any *lambdapure* file output directly from the Lean4 compiler, forming the transition between the presented MLIR compiler and the Lean interface. The Abstract syntax tree (AST) produced by the parser allows us to generate our

initial MLIR file with one-to-one translation *lambdapure* expressions to our *lambdapure* dialect operations as described in section 3.2.

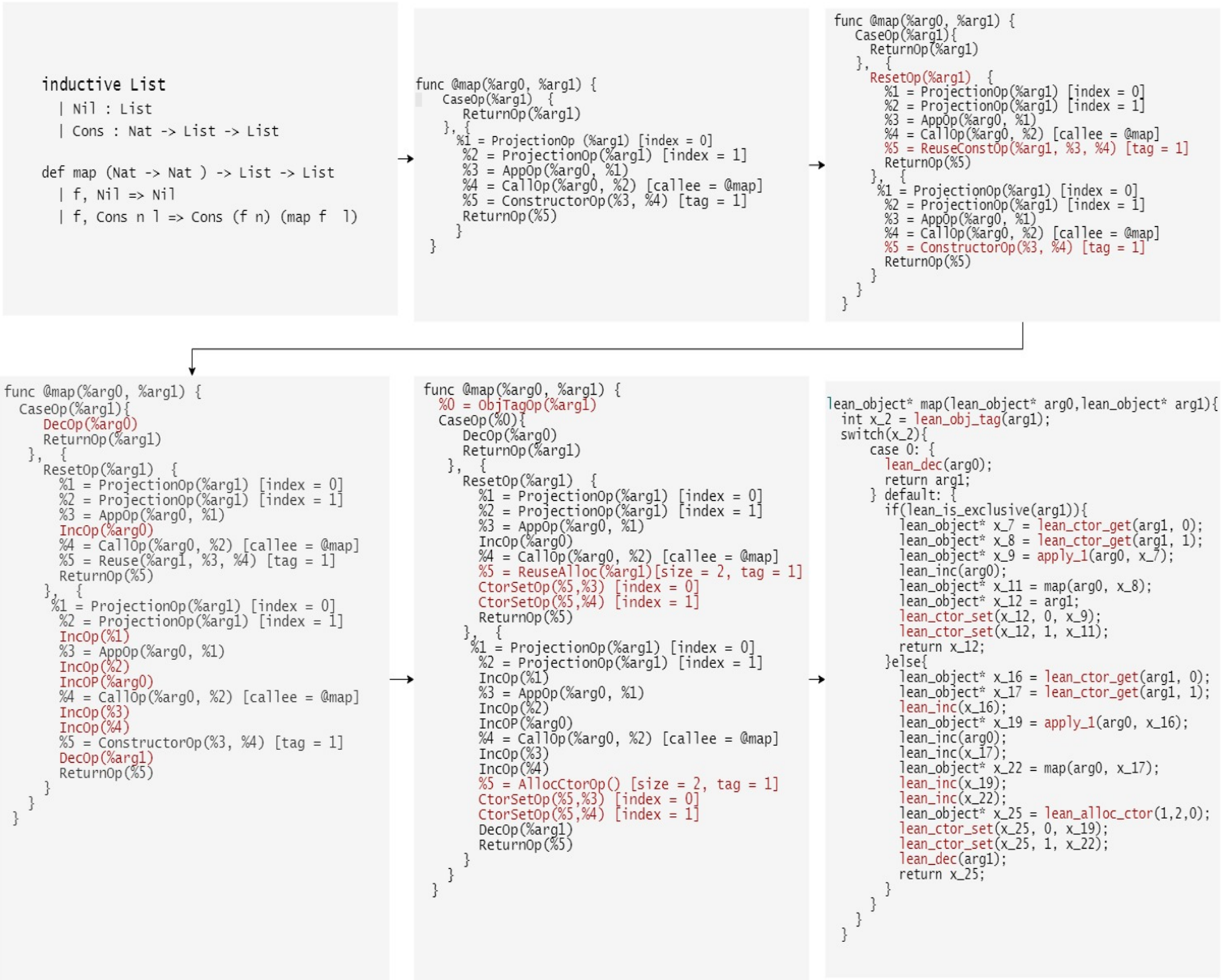

Figure 26: A full transformation of the map function going through all passes. The changes to the IR of each pass are marked red, and in the C code, the lean runtime invocations are marked red.

4 Evaluation

In this chapter, we examine performance results on a set of Lean programs, by comparing output code produced with and without applying the destructive updates pass. The metrics we measure are (1) the number of object allocation to the runtime environment that is made by the output C file, (2) the runtime of the binary result, (3) the maximum resident heap size, approximating the peak memory usage of the program. To model different programs benefiting from destructive updates, we wrote a handful of Lean programs as benchmarks. The results were checked for correctness with the current lean4 compiler. All performance measurements were made on an Intel i7-4770K CPU at 3.50 GHz and 16 Gb of RAM. We introduce the Lean programs used for the benchmarks:

- **Map**: The map function is a common example throughout this thesis and serves as the simplest example where the benefits of the destructive updates can be observed. The function takes a higher-order function and applies it to all elements of the list.
- **Filter**: The filter function checks all elements of a list with a condition and removes all items that do not fulfill the condition. We applied the filter function with a selectivity of 50 percent.
- **Swap**: The swap function swaps the right and left subtree of all nodes in a tree.
- **BubbleSort**: This function returns a sorted list and is an implementation of the common sorting algorithm BubbleSort.
- **BST:** The BST inserts elements into a binary tree, such that it is a valid binary search tree
- **Pair:** Pair represents the function updating a list of pairs into a list of single values, by adding both elements of the pairs together. This shows a situation where the destructive update replaces the reused object with a smaller object, causing memory fragmentation.

4.1 Allocation count

In figure 27 one can observe the difference of the number of allocation counts between the naive output and the output with the destructive updates. The map function with the destructive updates performs half the number of allocations, solely the allocations required to create the list. The destructive updates allow us to mutate the data inside the list without ever allocating new memory. Similar results are visualized in the filter and pair program. However, the largest difference is observed in the sort function. This shows the ideal application of a destructive update, where elements in a list are shuffled around many times throughout the sorting process. Moving to slightly more complex data-structures, the BST and Swap function also reduce allocation counts on a tree-like data-structure.

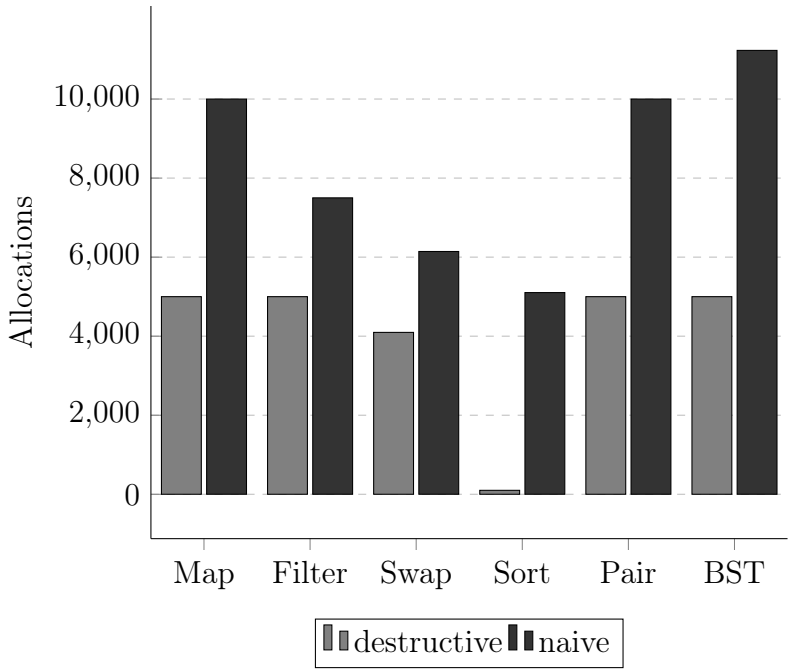

Figure 27: destructive updates pass yields a substantial decrease in the number of allocations made throughout the execution of the programs

4.2 Runtime

In this section, we observe how the reduced number of allocations affects the runtime of the benchmark programs. In figure 28 we see an increase in performance after the destructive updates pass on the map function. Essentially, paying the price of adding runtime checks to avoid allocations is a worthwhile investment in the case of map.

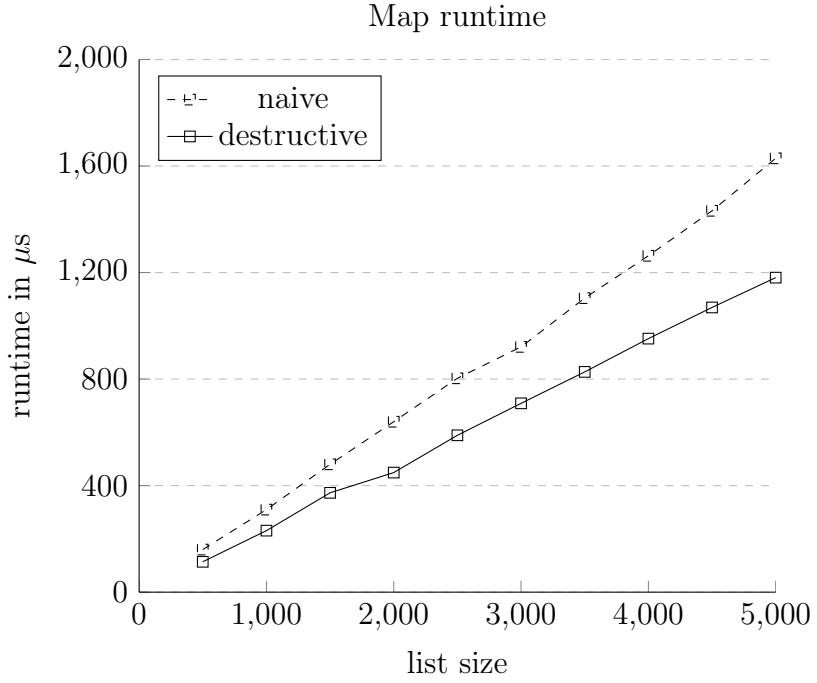

Figure 28: runtime of the map function before and after destructive updates shows a gain in runtime as a result of a reduced number of allocation calls

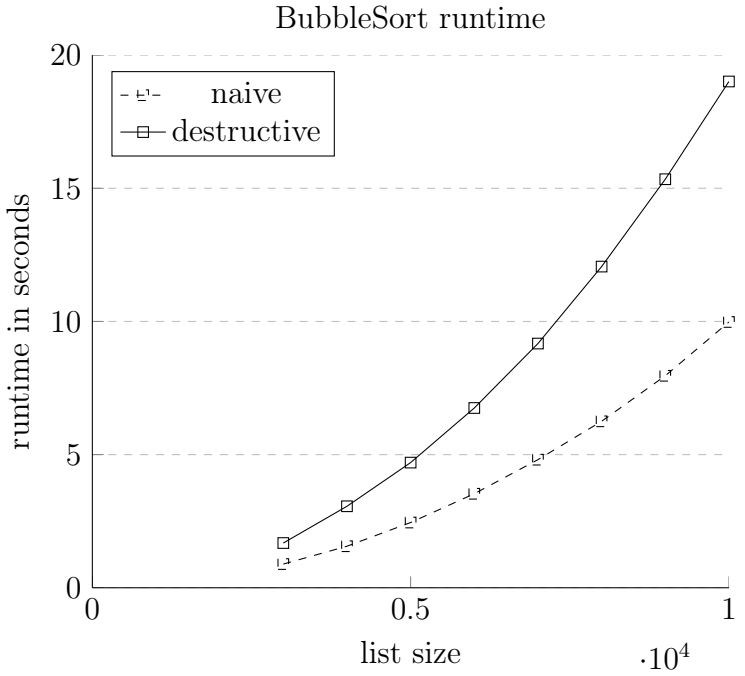

Figure 29: BubbleSort runtime with destructive updates is substantially better

The effects of the optimization are even larger in the visualized runtime on BubbleSort in figure 29. The difference in the number of allocations made is even larger than in our map function and further strengthens the case for destructive updates and the implied reduction of allocation calls. This implementation of sort represents an ideal case for destructive updates, where changes to unshared objects and the resurrection hypothesis occur frequently.

On the contrary, figure 30 averts expectations of previous examples and shows faster runtimes using the naive implementation. The filter function removes elements from the list that do not match a certain condition. With the naive implementation, the list data-structure is stored into memory newly allocated. On the other hand, with destructive updates, no new memory is allocated. The combination of reusing objects and removing elements in the list might cause the list layout in memory to lose spatial locality. The loss of spatial locality on top of runtime exclusivity checks might be more expensive than the performance gain from reusing objects. This remains merely a hypothesis and would require further in-depth analysis of the lean runtime system and implementation. Overall, these results suggest that destructive updates might not be a one-size-fits all optimization and not generally applicable to decrease runtime.

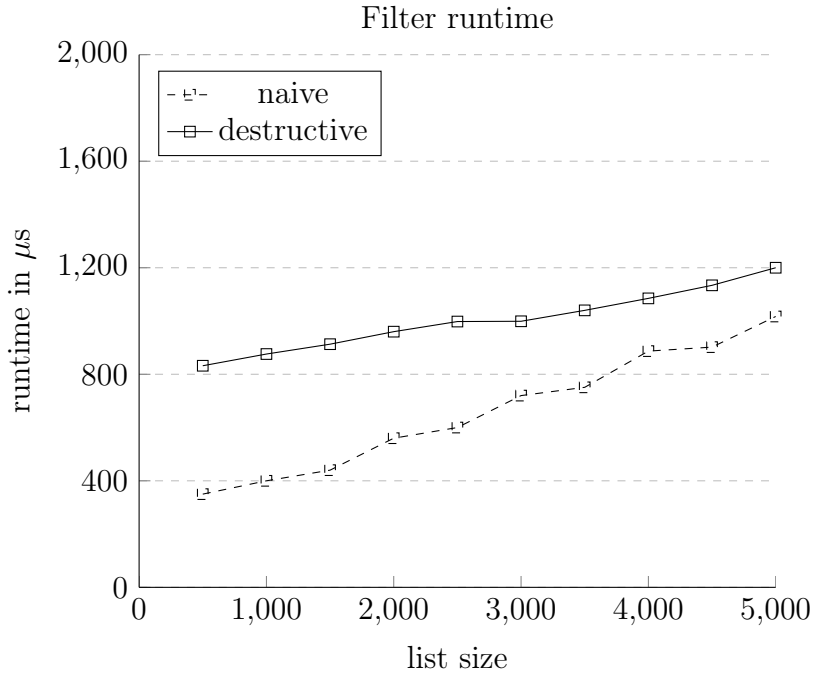

Figure 30: runtime of filter function before and after destructive updates shows faster runtimes without destructive updates

4.3 Peak memory usage

To perceive the influence of the reduction in allocation calls we observe the peak memory usage approximated by the maximum resident heap size during execution as shown in figure 31. As expected, the difference is negligible in all cases except for a small difference in the case of memory fragmentation during the destructive updates in the pair example. As a result the version with the destructive updates has a slightly higher peak memory usage than the naive implementation. This represents a trade-off between faster runtime and less memory usage.

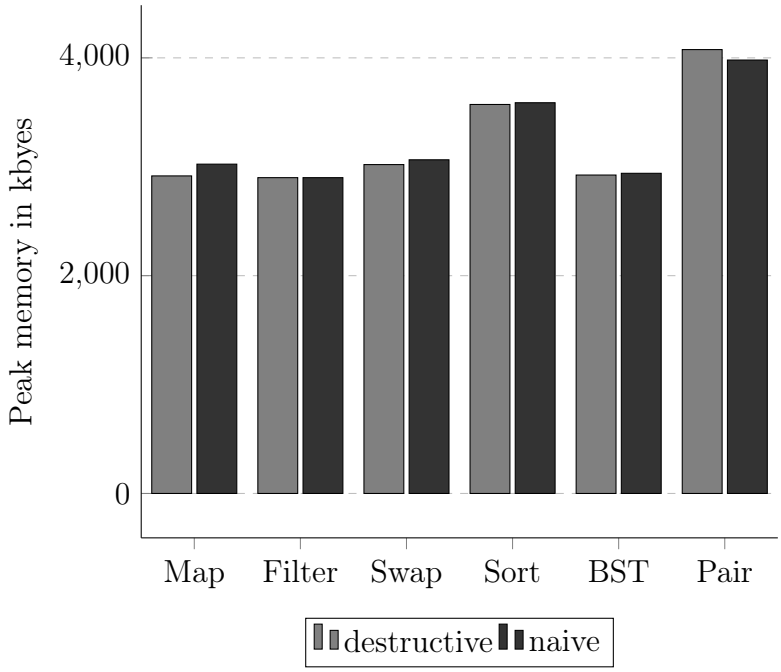

Figure 31: approximation of peak memory usage comparing destructive updates and naive implementation

5 Related Work

The idea of representing functional programming language in an SSA-based framework is also being explored by Siddharth Bhat. He is tackling the additional challenges of representing non-strict languages, specifically working on an MLIR dialect modeling the Haskell IR named Core [13, 4]. The transformations of destructive updates become more complicated to perform, due to the nature of laziness and the possibility of cyclic data structures. He pursues a strategy of performing demand analysis, which is unnecessary in the context of strict languages.

Representing functional programming languages in MLIR is being explored for machine learning workloads by Martin Lücke, Michel Streuwe, and Aaron Smith, developing RISE [14], a functional pattern-based language as a dialect in MLIR. Similar to *lambdapure*, the RISE dialect is based on lambda calculus and they introduce an MLIR operation *lambda*, wrapping a region to model a functional lambda expression. The resulting value returned by the *lambda* operation can be applied to a RISE data type for evaluation of the lambda expression. In their paper, they illustrate an example use of matrix operations represented in the RISE dialect. Furthermore, they intend to explore the integration of RISE with existing tensor-algebra optimizations in MLIR.

Reference counting is currently being examined in modern compiling systems. Although swift is not a purely functional language, it has implemented an intermediate representation with explicit reference counting instructions and inspired Lean to do the same. The experimental results show the competitiveness of reference counting memory management even when compared to garbage collection techniques [15, 16]. It will be interesting to observe the continued performance evaluations as reference counting will be further analyzed and optimized. Additionally, Sebastian Ullrich and Leonardo de Moura, the authors of *Counting Immutable beans*, are continuing to improve on the optimization of Lean and *lambdapure* and hope to provide *lambdapure* as a target for other purely functional programming languages.

Finally, destructive update inference techniques are examined by Martin Odersky without any runtime checks of exclusivity, but instead statically inferred using abstract interpretation on a small first-order single assignment language [17]. Static inference of possible destructive updates gets more complicated when applied to higher-order languages and the first-order language introduced is not able to capture the same semantics as the *lambdapure* intermediate representation. However, the ideas could help in eliminating runtime checks for simple cases.

6 Conclusion and future work

We presented a Lean compiler starting from the *lambdapure* IR to produce executable code using the MLIR framework. The destructive update optimization alongside the reference counting pass has a substantial decrease in memory allocations made during the execution of a small set of sample lean programs and a measurable decrease in runtime in some cases. An SSA based framework is shown to be suitable for modeling lambda calculus derived IRs with the addition of regions. This implies modeling lambda calculus in SSA can provide benefits to the world of functional programming compiler and in the future could allow for higher accessibility between different language representations. While this project has focused on a pure and eager functional language, we believe it can be extended to apply to non-pure and even lazy languages. Furthermore, having a large SSA support for lambda calculus derived intermediate representations could be beneficial to imperative style languages. As the line between functional and imperative programming paradigms continues to blur, we can imagine exploring integrating these types of optimization into the functional elements of imperative programming languages like Swift.

Explicit reference counting techniques are being explored with Swift and Lean showing promising performance evaluations. The reference counting insertions in our implementation could be optimized to reduce the number of counts by allowing for *borrowed references*. This can be beneficial in a function that rarely consumes a token at execution an increment and decrement are needlessly made every time the function is called. In summary, reducing the reference counts required throughout execution is an area for further research.

Focusing on the destructive updates optimization, there is still further work that can be done to capture a larger set of possible destructive updates. Our destructive updates technique relied solely on information gained through analyzing the operations inside a function. We determined potential candidates based on the usage of an object in other operations, such as inferring size based on projection operations or tags based on the location inside a case operation. While this works in many examples shown, we encountered a few where the optimizations could not be detected. We believe having a *lambdapure*-Esque dialect with more type information could provide more information and detect more destructive updates. Additionally, we could explore increasing the efficiency of destructive updates in situations the arguments of the new constructor are equivalent to the arguments of the new constructor, potentially saving a memory write.

Ultimately we wish to maintain high-level domain-specific information of the lambda calculus derived intermediate representations, but increase the accessibility of the optimizations between different implementations. The representation in an SSA system might ease the reusability of compiler infrastructure

among functional languages and the integration process of functional optimizations in languages where the line between functional and imperative continue to blur. We hope that this representation opens new avenues of thought and further increases performance in a world becoming increasingly functional.

7 Acknowledgments

I am extremely grateful to my supervisor, Prof. Dr. Tobias Grosser, for guiding me through writing my first thesis and providing helpful insights throughout the whole process. I would also like to thank Siddharth Bhat for discussing our projects together. Finally, I would like to thank Prof. Dr. Zhendong Su for allowing me to write this thesis.

References

[1] "lean4 compiler." https://github.com/leanprover/lean4. Accessed: 2020-11-03.

[2] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," 2020.

[3] S. Ullrich and L. de Moura, "Counting immutable beans: Reference counting optimized for purely functional programming," 2019.

[4] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler, "The glasgow haskell compiler: a technical overview," in *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, vol. 93, 1993.

[5] S. Dolan, L. White, and A. Madhavapeddy, "Multicore ocaml," in *OCaml Workshop*, vol. 2, 2014.

[6] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *International Conference on Automated Deduction*, pp. 378–388, Springer, 2015.

[7] L. d. M. S. K. Jeremy, Avigad, *Theorem proving in Lean*. 2014.

[8] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," *SIGPLAN Not.*, vol. 28, p. 237–247, June 1993.

[9] A. Kennedy, "Compiling with continuations, continued," in *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pp. 177–190, 2007.

[10] "MLIR language reference." https://mlir.llvm.org/docs/LangRef/. Accessed: 2020-11-02.

[11] M. Weingarten, "Lambdapure." https://github.com/mattweingarten/lambdapure, 2020. Accessed: 2020-11-02.

[12] "The lean runtime." https://github.com/mattweingarten/lean4/blob/init/src/runtime/lean.h. Accessed: 2020-3-18, forked from lean4.

[13] S. Bhat, "Core-MLIR." https://github.com/bollu/lz. Accessed: 2020-1-12.

[14] M. Lücke, M. Steuwer, and A. Smith13, "A functional pattern-based language in mlir,"

[15] L. de Moura, S. Kong, J. Avigad, F. V. Doorn, and J. von Raumer, "The lean theorem prover." 2015.

[16] Y. Wu, L. Li, S. Russell, and R. Bodik, "Swift: Compiled inference for probabilistic programming languages," 2016.

[17] M. Odersky, "How to make destructive updates less destructive," in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 25–36, 1991.