INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Designing a MLIR Dialect for the Brainfuck Language

Jacob Schwaiger

Draft vom November 22, 2023

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Designing a MLIR Dialect for the Brainfuck Language

Jacob Schwaiger

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Pascal Jungblut
Abgabetermin:	23. November 2023

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 23. November 2023

[illegible]

(Unterschrift des Kandidaten)

Abstract

MLIR is an extensible compiler infrastructure with support for parsing, printing, documentation, conversion and optimization of intermediate representation (IR). While being a rather young project, it is already used in applications ranging from representation of machine learning models, polyhedral optimization and quantum assembly languages. The extensibility of MLIR's IR is enabled by the concepts of operation (Op) and dialect. While Ops are a way to model functionality of any level, dialects can group operations of a common functionality or motivation.

The extensibility of MLIR makes it a rather complex framework. Users can extend the MLIR IR and pass infrastructure both declarative and imperative. Furthermore, they can use existing dialects and passes to combine them with their custom ones. To improve the accessibility of MLIR for the general public, we designed a starter project simple enough to focus on the possibilities of MLIR and complex enough to include the use of many of MLIR's features.

This thesis presents a MLIR-based compiler for the Brainfuck language: the MLIR Brainfuck project. We implemented frontend, middleware and backend based on an import script to translate Brainfuck programs to an MLIR representation, three MLIR inherent abstractions to model Brainfuck on different levels, conversion passes between them and a conversion to the LLVM intermediate representation.

Contents

1 Introduction

MLIR is a compiler framework inspired by LLVM integrating core compiler tasks such as parsing, printing, documentation, conversion and optimization of an intermediate representation (IR) while being fully extensible in terms of IR constructs and ways to operate on it (passes)[LAB+21]. As a consequence, MLIR is used in the realms of machine learning [JBL+20], polyhedral optimization [MCZZ21] and quantum assembly languages [MN21]. Despite the differences in the subjects, they require representing concepts of different abstractions and exploitation of an existing compiler infrastructure.

MLIR builds on concepts to enable extending the IR with constructs of different abstraction levels while still allowing them to coexist in a combined IR. The fundamental concept of MLIR is an *operation* (Op) [LAB+21]. Ops are used to represent constructs of any level of abstraction (module, function, instruction). Operations can be grouped by functionality to form *dialects* [LAB+21]. Core or custom dialects are integrated into the MLIR repository and can be used [noa23k]. This approach makes it possible to implement frontend (parsing, higher-level-representation), middleware (optimizations on different levels of abstraction) and backend (generation and optimization of machine code) of a compiler in MLIR by combining compositions of existing or self-defined dialects. Since users should be able to add new constructs in a quick and standardized way, MLIR integrated a declarative TableGen [noa23a] based infrastructure to declare and partially define constructs and generate the required Cpp code automatically. Some constructs require additional definition trough the Cpp API [noa23g, noa23x].

Unfortunately, the extensibility of MLIR makes it a rather complex framework. To get started users have to understand the MLIR IR, the existing dialects, the pass framework and the declarative and imperative infrastructures to extend MLIR and how they work together. An experimental starter project could help improving accessibility.

The MLIR Brainfuck project aims to experiment with the MLIR infrastructure to provide a compiler for a simple programming language while exploiting the framework as much as possible. As a language, we chose the Brainfuck language [noa23e], a minimalist but Turing-complete programming language.

We implemented a MLIR dialect to model Brainfuck at the highest level, merged further custom and existing dialects to represent Brainfuck in three levels of abstraction and lowered the language to the LLVM intermediate representation. The custom dialects are declared and defined declaratively with the help of the Operation Definition Specification (ODS) [noa23w]. The conversion passes used in the lowering steps are declared and partially defined declarative, but for the most part, implemented directly using the Cpp API of MLIR.

The thesis is structured as follows: After an overview of the Brainfuck language in section 2, we present core principles of compiler design and a short description of the LLVM compiler framework in section 3. Section 4 gives an overview of the MLIR compiler framework. The core part of the thesis is the presentation, evaluation and discussion of the MLIR Brainfuck project in sections 5, 6 and 8 respectively.

2 The Brainfuck Language

The Brainfuck language is a minimalist programming language invented by Urban Mueller [noa23e]. It only consists of eight commands. The textual representation of each command is one character. This enables small program files and compilers. Brainfuck is proven to be Turing-complete.[Mor15]

2.1 Specification

A Brainfuck program uses eight commands to operate on a linear arrangement of cells called *memory*. Through a *pointer* each cell can be accessed. We refer to the cell accessible by the pointer at a given moment in execution time as the *current cell* (cc).

The memory length and storage size of the cells are not specified. Different implementations use different memory and cell sizes, leading to different languages. Note that the storage size of the pointer must be at least as large as the memory length.[noa23e]

We give the syntax and semantics of each Brainfuck command respectively.

Command	**Effect**
$>$	Shifts pointer to the right by one cell
$<$	Shifts pointer to the left by one cell
$+$	Increases the current cell
$-$	Decreases the current cell
.	Outputs the current cell
,	Inputs a new value for the current cell
[	If cc is zero, moves pointer to matching]
]	If cc is not zero, moves pointer to matching [

The 'Command' column contains all characters that Brainfuck recognizes; all other characters are ignored [noa23e]. Since the command '`[`' matches the command '`]`' iff the current cell is not zero, Brainfuck code contained in the brackets is executed as long as $cc \neq 0$; we will refer to this condition as the *Brainfuck loop condition*.

2.2 Brainfuck Configuration

In the following, we assume that the memory contains 30000 cells, that each cell can store one byte and is wrapped accordingly and that the cell values are input and output as ASCII values. We ensure all cells are set to zero and the pointer is set to the first cell at execution start. This reflects the settings of Urban Mueller's reference Brainfuck compiler [26223b].

2.3 Properties

We list properties of the Brainfuck Language either because they are important for its implementation (Syntactic Correctness, Instruction Classification, Abstractions, C Language Representation) or to justify the MLIR Brainfuck project (C Language Representation, Computational Class).

2.3.1 Syntactic Correctness

A Brainfuck program is syntactically correct if it contains an even number of brackets. For each occurrence of '`[`' there has to be an occurrence of '`]`'.

2.3.2 Instruction Classification

Given that a Brainfuck program is syntactically correct, the commands '`[`' and '`]`' form a loop with respect to the Brainfuck loop condition.

The Brainfuck commands can be divided into classes. The instructions '`>`', '`<`', '`+`', '`-`', '`.`' and '`,`' form the class of *primitive* commands. The instructions '`[`' and '`]`' form the class of *context inducing* commands. This is because an executing device must memorize the amount of opened and closed brackets to ensure correct execution.

Since the semantics of a loop are induced by the instructions it contains, it is also justified to speak of the second class of instructions as *composed* instructions.

An implicit composed Brainfuck construct is a Brainfuck module given by Brainfuck code stored in a file; we represent this as (code, file).

2.3.3 Abstractions

The Brainfuck language depends on memory, pointer and console. The memory is used to store the cells targeted by the primitive commands. The pointer is used to address the current cell. The console is needed to provide input and show output. All three constructs are not directly accessible to the users. Once the memory size and pointer width are defined, the language can be specified without giving details on their implementation. Implementations have to find ways to represent the memory, pointer and console accordingly. The parameters to be reflected during implementation will depend on the properties of Brainfuck and the properties of the framework used for implementation.

2.3.4 C Language Representation

We can give a C langauge representation of the Brainfuck language. Listing 2.3.4 provides such an implementation regarding our specified configuration 2.2. The following listing gives a mapping between Brainfuck commands and their C language representation. Note that

- the first two mappings abbreviate Brainfuck code with '..',
- assuming syntactic correctness, we represent `[` and `]` as one command,
- the implicit Brainfuck abstractions memory, pointer and console are *explicit* in the C implementation.

```
( .. , .bf) =>          int mem[30000] = { 0 };
                        int p = 0;
                        int main() { .. }
[ .. ]      =>          while (mem[p] != 0) { .. }
+           =>          ++mem[p];
-           =>          --mem[p];
>           =>          ++p;
<           =>          --p;
.           =>          putchar(mem[p]);
,           =>          mem[p] = getchar();
```

Listing 2.1: C Language Representation of Brainfuck.

2.3.5 Computational Class

The Brainfuck language has been shown to be Turing-complete under idealizations [noa23e]. An example of a proof using the idealization of an unbounded memory and a bounded cell size is Daniel B. Cristofani's [Cri23] implementation of a (universal) Turing machine in Brainfuck, which proves the Turing-completeness by simulation [noa23e]. The ratio behind a proof by simulation is the observation that if, for given languages P and Q, an interpreter for P can be written in Q, then Q can solve as many problems as P [noa23h]. Since in our case $P \mapsto TuringMachine$ and $Q \mapsto Brainfuck$ this results in Brainfuck being Turing-complete.

2.4 Examples

We present some example code to get acquainted with the Brainfuck language.

`[ .. ]` Any Brainfuck program of the form `[ .. ]` does nothing: since the cells are set to zero at execution start, the loop is not executed or, in other words, the opening bracket matches the closing bracket immediately.

`+` The idiom `+` (`-`) increments (decrements) the current cell. Any sequence of n incrementations (decrementations) can be read as

$$current\ cell = current\ cell + n$$
$$(current\ cell = current\ cell - n)$$

which is of course not a Brainfuck expression and can only be expressed in Brainfuck via the associated n sequence. We could give a C language implementation of Brainfuck that can express the shortened version.

`[-]` The idiom `[-]` (`[+]`) can be used to set the current cell to zero. The loop is executed until the current cell is equal to zero. Since the 8-bit cells are wrapped, once the minimal (maximal) value that can be represented in 8-bits is reached, any further decrement (increment) would flip all bits to zero. Then the loop terminates. This means that after loop execution the current cell is always zero.

`<` The idiom `<` (`>`) shifts the current cell to the left (right). Note, that in the Brainfuck configuration 2.2 `<` at execution start is illegal, since it leaves the Brainfuck memory. Any sequence of n left (right) shifts can be read as

$$\begin{gathered} pointer = pointer - n \\ (pointer = pointer + n) \end{gathered}$$

which is of course not a Brainfuck expression and can only be expressed in Brainfuck via the associated n sequence. Again, we could give a C language implementation of Brainfuck that can express the shortened version.

`> .. <` The idiom `> .. <` ends at the same cell as it started since the commands `>` and `<` cancel each other, assuming that the intermediate Brainfuck code contains either no shifting commands or only occurrences of the idiom.

`+[>,.<]` The idiom `+[>,.<]` is a never terminating feedback loop. The program starts with incrementing the current cell; we fix this cell as c_0. Therefore, the following loop will execute at least one time. We observe that the loop body matches the previous idiom. This means that the opening bracket and the closing bracket always check against c_0. Accordingly, the loop never terminates, since c_0 is never zero. Therefore, the loop body '`,.`' inside the previous idiom gets executed 'forever'; it inputs a new value and then outputs it again.

3 Theoretical Foundations

3.1 Compilers

Compilers are programs that convert a *source* language to a *target* language. Often the target language is the machine language.[WG84, ULSA08]

Compiler Pipeline Let us fix some source language $\mathcal{L}$ and a program P written in $\mathcal{L}$. P is input to a compiler $\mathbf{C}_{\mathcal{L}}$ as a stream of signs. Zooming into $\mathbf{C}_{\mathcal{L}}$, which tasks are found? A classical pipeline [ULSA08] consists of a

1. Lexer: Groups the incoming signs to tokens defined by $\mathcal{L}$,
2. Syntactical Analyzer: Transforms the token stream to a syntax tree that corresponds to the rules of composition of $\mathcal{L}$ and stores symbols in a symbol table,
3. Semantical Analyzer: Uses the syntax tree and the symbols to verify that the semantics of P are consistent with the language definition,
4. Inherent Code Generator: Uses the syntax tree and the symbol table to generate compiler inherent code,
5. Inherent Code Optimizer: Uses the compiler inherent code to perform optimizations,
6. Machine Code Generator: Uses the compiler's inherent, optimized code to generate machine code
7. Machine Code Optimizer: Optimizes the generated machine code.

We can further classify the components involved in the pipeline. We will refer to the lexer, syntactical analyzer and semantical analyzer as the *frontend*. The pair of inherent code generator and optimizer we will call *middleware*. Additionally, we call the inherent compiler code *intermediate representation* and abbreviate it with IR. Note that steps four and five are repeatable since a compiler can specify IR of different abstractions to enable optimizations that require different levels of abstraction. The pair of machine code generator and optimizer is labelled as *backend*.

Dependency of Compiler Components We characterize the dependencies of frontend, middleware and backend; they can be dependent on the source language and on the machine language.

The frontend is dependent on the source language since the steps involved require knowledge about its specification: the specification declares the language tokens, their rules of composition and their semantics.

The middleware can be dependent on the source language. Some language-specific constructs need special representation in order to be optimized. Still, abstractions to more than one source language are possible: since programming languages share common constructs, there are common ways to represent them in a way that can be optimized.

The backend is machine code dependent: it targets a specific machine-related instruction set. It seems plausible to abstract machine-related instruction sets and outsource specific machine code generation.

Passes It is possible to group steps *functionally.* We will refer to a step grouping that expects an input object i and returns a specified product o as *pass*; we can represent a pass with the notation $i \rightarrow o$.

An example is the grouping of steps one to four under the specification $P \rightarrow \mathbf{C}_{\mathcal{L}}(IR)$. This pass is a frontend pass. Accordingly, a backend pass is given by $\mathbf{C}_{\mathcal{L}}(IR) \mapsto machine(IR)$. [ULSA08] A pass associated with steps four and five takes the IR and returns some optimized form of it; these passes are referred to as *optimization* passes. If a compiler has multiple IRs, it requires transformations between them; passes that perform a mapping $IR_1 \rightarrow IR_2$ are called *conversion* passes.

3.2 Intermediate Representation of Data and Flow

A compiler IR must concisely present data to enable simple data flow analysis for optimizations. One approach is the Single Static Assignment (SSA) form [CFR+91]. A program is in SSA form if each statement evaluates an expression and either assigns the result to some variables or determines a branch.

An assignment consists of a n tuple of variables at the left-hand side and a matching n tuple of expressions at the right-hand side separated by some sign designating an assignment. Note that each entry $i \in \{1, .., n\}$ at the right-hand side matches the entry $j \in \{1, .., n\}$ at the left-hand side with $i = j$. Further, note that each variable is unique since it is the target of exactly one assignment.[CFR+91]

In the following listing, each occurrence of V on the left-hand side is replaced by a unique variable. The Variable `V_1` is only reused on the left-hand side.

```
            V = 1                   =>              V_1 = 1
            V = V + 1               =>              V_2 = V_1 + 1
            V = 3                   =>              V_3 = 3
```

Listing 3.1: SSA Assignment.

To extract variables from the branching context a ϕ function is used: Each path in a branching context results in a SSA variable; the ϕ function encodes which assignments reach the join point.[CFR+91]

```
1           if c                    =>              if c
2               then V = 1          =>                  then V_1 = 1
3               else V = 2          =>                  else V_2 = 2
4                                   =>              V_3 = phi(V_1, V_2)
5           /* V is used */                         /* V_3 is used */
```

Listing 3.2: SSA Branching.

Since in SSA form each assignment results in a unique variable, the *else* and the *then* path result in distinct variables. They are joined in line four and the application of the ϕ function assigns the result of the branching.

3.3 Compiler Frameworks

Programming languages typically need either a compiler or an interpreter to enable program execution. Here we focus on compiled languages. As we have seen in paragraph 3.1, the implementation of a compiler is rather complex. On the other hand, we identified potential for abstractions beyond a given source language. *Compiler frameworks* present users with an infrastructure to write their compilers in. For example, they provide a specific IR that users can target. Given an IR, the compiler framework can specify optimization passes and backend passes to lower it to machine code. With this approach, the user only has to implement a frontend pass.

3.4 LLVM

An example of a compiler framework that specifies one IR, optimization passes on it and backend passes for different targets is LLVM [LA04, noa23r]. We briefly describe the LLVM intermediate representation, and the LLVM compiler architecture and identify limitations of the framework.

3.4.1 LLVM Program Representation

The code representation of LLVM (LLVM IR) is aimed to provide enough high-level information to enable analyses and transformations. At the same time, it should be low-level enough to represent arbitrary programs and permit far-reaching optimizations.[LA04]

It consists of an instruction set, primitive types (Boolean, integer (8 - 64 bit) and floating-point (single and double precision)), derived types (pointer, arrays, structures and functions), and an infinite set of typed virtual registers. The virtual registers are in SSA form. This implies that each virtual register is written to only once [LA04].

The instruction set is based on key operations of processors while avoiding machine-specific constraints (physical registers, pipelines, low-level calling conventions). It includes arithmetic and logical instructions as well as instructions for memory allocation, memory access (loading, storing) and casting. Additionally, branching instructions are used to model control flow.[LA04]

Memory usage is explicit and typed in LLVM: memory allocation, reference and access are modelled using typed pointers. Also, functions and global variables are referenced using pointers.[LA04]

LLVM IR is organized into functions, blocks and instructions. Each block ends with a terminator instruction that has an attached successor block. Branched blocks are merged with the `phi` instruction to extract branching results.[LA04]

Listing 3.4.1 shows the implementation of the feedback loop example 2.4 in LLVM IR. Note that, in analogy to the C implementation of Brainfuck, the implementation has explicit representations of the Brainfuck memory, pointer and console (lines one to four). LLVM memory reference and access are performed on the pointer (f.e. line 10) and the memory (f.e. lines 11-12). The LLVM block and branching concept enables the representation of the loop: while the block with label `5` implements the Brainfuck loop condition, the block with label `10` implements the loop workload. Block `10` terminates with a branching instruction that specifies block `5` as successor. Block `5` terminates with a conditional branching instruction depending on the result of the Brainfuck loop condition (calculated in line 19): if it is true control flow is passed to block 10, else control flow is passed to block 20 and the loop terminates.

```
@bf_ptr = private global i64 0
@bf_memory = private global [30000 x i8] undef
declare i32 @getchar()
declare i32 @putchar(i32)
declare ptr @malloc(i64)
declare void @free(ptr)
define void @main() {
  %1 = call i32 @getchar()
  %2 = trunc i32 %1 to i8
  %3 = load i64, ptr @bf_ptr, align 4
  %4 = getelementptr i8, ptr @bf_memory, i64 %3
  store i8 %2, ptr %4, align 1
  br label %5

5:                                                ; preds = %10, %0
  %6 = load i64, ptr @bf_ptr, align 4
  %7 = getelementptr i8, ptr @bf_memory, i64 %6
  %8 = load i8, ptr %7, align 1
  %9 = icmp ne i8 %8, 0
  br i1 %9, label %10, label %20

10:                                               ; preds = %5
  %11 = load i64, ptr @bf_ptr, align 4
  %12 = getelementptr i8, ptr @bf_memory, i64 %11
  %13 = load i8, ptr %12, align 1
  %14 = sext i8 %13 to i32
  %15 = call i32 @putchar(i32 %14)
  %16 = call i32 @getchar()
  %17 = trunc i32 %16 to i8
  %18 = load i64, ptr @bf_ptr, align 4
  %19 = getelementptr i8, ptr @bf_memory, i64 %18
  store i8 %17, ptr %19, align 1
  br label %5

20:                                               ; preds = %5
  call void @free(ptr inttoptr (i64 3735928559 to ptr))
  ret void
}
```

Listing 3.3: Implementation of Example 2.4 in LLVM IR.

3.4.2 Compiler Architecture

In the following, we present the LLVM compiler architecture. The key point, from a user's perspective, is that the optimizing capabilities of LLVM are the consequence of targeting the LLVM IR.

Frontend Compilers that target LLVM IR have to implement lexer, syntactical and semantic analyzers specific to the source language resulting in a language-dependent intermediate representation (LD IR). It is optional to perform language-specific optimizations on this representation. The mandatory task is to convert LD IR to LLVM IR while synthesizing as much LLVM type information as possible to enable LLVM-based analysis and transformation. As a last, optional step, LLVM passes can be performed on the module-level of the generated LLVM IR for global or inter-procedural optimizations. [LA04]

Linkage and Global Optimization From this point on, the LLVM infrastructure takes over. At link time aggressive interprocedural optimizations across the entire program are performed on the LLVM IR; see [LA04] for details on the various optimization passes included in LLVM.

Native Code Generation Native code for the target platform is generated after linkage and LLVM IR-specific optimizations. Here runtime optimizations are prepared by introducing instrumentation to identify frequently executed code regions. [LA04]

Runtime Optimization During runtime frequently executed regions of code are identified using the generated instrumentation. In this region, a runtime instrumentation library analyses the executing native code to identify frequently executed paths. The original LLVM IR associated with the path is duplicated, LLVM optimizations are performed on it and native code is regenerated. [LA04]

3.4.3 Limitations

LLVM does not specify a universal compiler IR [LA04]. Rather, the LLVM compiler structure outsources compiler implementation tasks five to seven but does this using a rather low-level program representation as a starting point. The program representation is static in the sense that users have to fit the language-dependent types and features into it [LA04] Therefore, frontends have to provide language-dependent representation and optimization for features that can not be represented in the LLVM IR or for representations that would otherwise result in information loss preventing optimization. In summary, the core problem is the lack of a notion of extensibility: the instruction set and types are fixed and the provided passes are dependent on these components.

4 MLIR

MLIR is a compiler infrastructure with a focus on extensibility and modularity. It is a reaction to fragmentation in the field of compiler design: while single-layered IR frameworks like LLVM are still widely used, many compiler systems introduce specific higher-level IR before targeting a single-layered IR framework. This leads to the reproduction of similar but still rather complex code; varying quality can be a consequence.[LAB+21]

MLIR aims to solve this problem by making it easy to define custom constructs of different levels of abstraction. Its IR is fully extensible: users can add *operations*, *types* and *attributes* under a common namespace to extend the IR to their needs; such a grouping of constructs with common functionality is called a *dialect*. Dialects can target any level of abstraction. Existing MLIR dialects range from frontend (linalg or tensor), middleware or common purpose (memref, scf or arith) to backend (llvm or SPIR – V) dialects [noa23k].

Users can implement all seven compiler design steps in MLIR. There is infrastructure to provide documentation and the needed parser and printing logic for the defined constructs. These are tasks we associated with a compiler frontend. MLIR provides general transformation, conversion passes between dialects and dialect specific passes. Additionally, the pass infrastructure can be extended with passes to operate on custom or already existing dialects. Together with the existing and custom dialects, the pass framework enables MLIR to support multi-level middleware. The llvm dialect provides operations, types and attributes to model the LLVM IR in terms of MLIR. Additionally, MLIR provides a way to translate IR composed of constructs of the lllvm dialect to LLVM IR. This is an example of a backend implemented in MLIR.

In the following, we give an overview of the IR design and the pass framework of MLIR respectively. We focus on concepts that are applied by the MLIR Brainfuck project and provide the reader with a perspective on further ways to use MLIR.

4.1 IR Design

[1] MLIR depends on the textual representation of its concepts. Since MLIR is extensible it needs a meta-definition for specifying the textual form of a newly defined construct. The MLIR Language Reference describes such a grammar [noa23u]; we refer to this specification as the *generic textual representation*.

The generic textual representation fully reflects the in-memory representation of the defined constructs. Users can define custom syntax and reduce verbosity in cases where information is empty or derivable.[LAB+21] During this section, we will use the generic textual representation. In cases where constructs of the MLIR Brainfuck project defined in section 5 use custom syntax, this is stated explicitly.

[1]The concepts presented in the following rely on each other. Sometimes the name of a concept is used before the concept is explained.

```
// Operation associated with dialect d_1 with m arguments, one attribute, n
// results and an attached region
%res_1,.., res_n = "d_1.op_1" (%arg_1,..,arg_m) ({ // Region
    ^entry_block(%arg: d_1.type_1): // Block
        // Operation associated with dialect d_2 with one result and a region
        %v = "d_2.op" ({ // Region
            "d_2.op_2"(): () -> ()
        }) : () -> !d_2.type_3
        "d_1.consume_value"(%v) : (d_2.type_3) -> ()
    ^term_block: // Block
        "d_1.terminator"() [^block(arg: !d_1.type)] : () -> ()
}) <{ attr = "value": !d_1.some_type}> : (!d_1.type_1,..!d_2.type_m) -> (!d.
    type_1,..,!d.type_n)
```

Listing 4.1: MLIR Intermediate Representation.

4.1.1 Operations

In MLIR an operation (Op) is a first-class object. It forms the unit of semantics [LAB$^+$21]. Ops capture constructs of different levels from "modules" and "functions" to "control flow statements" and "instructions". There is not a definite set of operations. Rather, MLIR is operation extensible. Users define custom operations using a declarative syntax based on TableGen [noa23a] called the Operation Definition Specification (ODS) [noa23w] or by interacting directly with the Cpp API.

Internally, operations are represented by the following data: they are identified by a unique name (the opcode), take optional operands, have optional results, have an optional dictionary of attributes, specify optional successor blocks and have optional enclosed regions [LAB$^+$21, noa23u]. The operands and results of an operation are maintained in SSA form [LAB$^+$21]. The generic textual representation captures all this information; see 4.1. If an operation is attached to a dialect, a custom assembly form can be defined [noa23u].

The data attached to an operation forms its invariants: after application of a pass to an operation, the invariants (number of operands and their types, the types of the results) should be the same. Additional invariants can be added by enriching the operation definition with traits and interfaces [LAB$^+$21, noa23w].

4.1.2 Regions and Blocks

MLIR uses regions to support nested structures. A region is attached to an operation and contains a least one block (the *entry* block). A block again consists of operations.[LAB$^+$21] The semantics of a region are determined by the operands and results of the enclosing operation [LAB$^+$21, noa23u].

MLIR specifies two kinds of regions: SSACFG regions, which do require control flow between blocks and Graph regions, which do not require control flow between blocks [noa23u]. Since MLIR Brainfuck does not rely on Graph regions, we assume all regions to be SSACFG. The operations in a SSACFG region are executed sequentially. Therefore, before an operation can execute, its operands must have well-defined values. After execution, the operands have the same and the results also have well-defined values.[noa23t] In SSACFG regions, to enable control flow [noa23u] blocks end with a *terminator* operation. The terminator operation can specify a *successor* block. As required in a nested context, terminator operations can transfer control flow to blocks contained in the same region or return it to the operation that encloses it [LAB+21].[2]

MLIR models data flow between blocks through block arguments. Block arguments are lists of types that correspond to SSA values [LAB+21]. The arguments of entry blocks are the arguments to a region which are the arguments to the enclosing operation. Block arguments of a non-entry block b are determined by the terminator operations that specify b as successor.[noa23u] This form of data flow is a functional form of SSA [App98] that does not use ϕ nodes explicitly [LAB+21].

4.1.3 Attributes and Types

Attributes provide information about operations at compile-time. Each operation has a key-value dictionary mapping of strings to typed values. The generic textual representation of attributes is a list of comma-separated key = value : type pairs enclosed in curly braces; see 4.1 line 12.[LAB+21]

Values in MLIR are typed. Type information is located in operation results or block arguments. In the generic textual representation, n input and m output types form a n and a m tuple separated by $\rightarrow$; see for example 4.1 line four and 12.[LAB+21]

4.1.4 Dialects

Dialects enable MLIR to be extensible in a modular way. They provide a grouping mechanism for operations, attributes and types under a common, unique namespace. The semantics of a dialect result from the Ops, attributes and types it contains; but these constituents need the dialect as a grouping mechanism to ensure their modularity. The generic textual representation reflects this by using the name of the dialect as a dot-separated prefix to the opcode of an operation; see 4.1.[LAB+21, noa23i]

4.1.5 Symbols and Symbol Tables

An operation can have an attached symbol name. This symbol name is stored in a symbol table and refers to the operation that defines the symbol: the symbol table is a mapping between names and IR. Symbols can be used to define and call functions, create named modules and declare global variables.[LAB+21]

Operations that *define* a symbol must use the SymbolOpInterface to provide verification and accessors [sym23].

Operations that *expect* symbols to be defined in their associated region must implement the SymbolTable trait. Adding the trait provides the operation with a container to store the mapping from symbol names to operations and adds verifying (f.e. unique names) and managing behaviour (f.e. easy lookup).[sym23]

Operations that *use* symbols need to implement the SymbolUserOpInterface interface. This adds verifying behaviour to the operation: they have to implement a check that the symbol they reference exists. Symbols are referred to using a SymbolRefAttr attribute. This attribute contains a named reference to the related operation that is stored in the symbol table.[sym23]

[2] Extra region control flow transfer is always defined: The enclosing operation must have a region attached that contains at least one block since otherwise the terminator operation could not occur in a nested context

4.2 Passes and the Conversion Framework

Transformations target operations because they are the core unit of abstraction. Since passes rarely need to know the details of an individual operation, MLIR provides *traits* and *interfaces* as a mechanism to enable generic passes [LAB+21].

4.2.1 Traits and Interfaces

Traits are a mechanism to group attributes, operations and types by implementation details and properties. They represent special properties or constraints and attach them to an object.[tra23] This informs transforming passes what kind of transformations are allowed on a given op; for example, the Commutative trait allows for swapping the operands of an operation [LAB+21, tra23].

Interfaces allow a (partial) view into the behaviour of an IR object, without making assumptions about implementation details. If an object implements an interface, it implements the action that should be performed locally on it by a pass. As a result, passes know both which objects are valid targets and how the general action should be performed on the specific object. The interfaces SymbolOpInterface and SymbolUserOpInterface [noa23p] are used by verification passes and require the implementation of verification hooks.

4.2.2 Pass Infrastructure

The pass infrastructure of MLIR provides general transformation, conversion and dialect specific optimization passes. A pass is created via inheritance of the OperationPass class and implementation of the virtual void runOperation() method. Passes do have access to the MLIRContext object. Before the application of a pass is possible all dialects that it depends on must be loaded to this context.

4.2.3 Conversion Framework

The MLIR Brainfuck project relies heavily on *conversion* passes. Conversion passes convert an operation into one or more operations within or between dialects. The conversion framework [noa23j] is used to implement and run conversion passes. The framework relies on *conversion targets*, *rewrite patterns* and optional *type converters* to run a conversion pass.

In the context of conversion, dialects and operations can be marked as *legal* or *illegal*. A legal operation can be contained in the result of a conversion pass; an illegal operation must not. If a dialect is marked as illegal (legal), we interpret all of the operations in a dialect as illegal (legal). An operation attached to a dialect that has been can be marked as legal (illegal), still may be marked as illegal (legal).

A conversion target specifies legal and illegal dialects and operations. The legal components are the building blocks of the conversion, while for each illegal operation, one rewrite pattern needs to be specified. However, a rewrite pattern can turn one illegal operation into a composition of multiple legal operations. Furthermore, the conversion framework can detect transitive relations between rewrite patterns allowing users to compose conversion of operations automatically [noa23j]

Type converters are used to reconcile type differences that can occur during conversion. One use case is the direct conversion between types [noa23j]. Take, for example, an illegal dialect defining a Boolean type and a legal dialect defining instead an Integer type that can be parameterized by bit width. This would require a type conversion $Boolean \mapsto Integer(1)$.

4.3 An Overview of Used Dialects

The MLIR Brainfuck project utilizes multiple dialects of the MLIR ecosystem. In the following, we describe each of them briefly.

builtin The builtin dialect is implicitly loaded to every MLIRContext and therefore directly available to all MLIR users. It defines a set of operations, attributes, and types that is widely applicable. For example, the IntegerType, MemRefType, IndexType and the builtin.module operation are defined by builtin. The builtin.module operation implements the SymbolTable trait. As a consequence symbol operations can be declared in the region attached to the operation. The IntegerType represent arbitrary precision integers with a fixed limit. Optionally an integer type can have signedness semantics [noa23v]. As an example, `i8` expresses an eight-bit sign-less integer, while `si32` denotes a signed 32-bit integer. The MemRefType models a structured multi-index pointer into memory; the expression `memref<30000xi8>` denotes a 30000 dimensional array of sign-less 8-bit integers. The index type encodes an integer-like type with a platform-dependent bit. This means that an index value has a symbolic width equal to the machine word size [noa23v]; it is equivalent to intptr_t in C [MCZZ21]. Furthermore, the builtin dialect specifies integer and index attributes. They store values known at compile time. The integer attributes reflect different bit widths.

func The func dialect provides operations to represent the declaration, definition and application of functions. The func.func operation is a go-to construct to describe control-flow dependent IR since it has an attached SSACFG region. [noa23m]

scf The scf dialect describes higher level control flow constructs. Instead of branching operations, it defines operations to model "if-then-else", "for-loop" and "while-loop" constructs. The scf.while operation can be used to represent both "while" and "do-while" loops. [noa23y]

arith The arith dialect holds integer and floating point operations. This includes operations to represent arithmetic, comparison, bit-wise and shift and casting ops. [noa23d]

index The index dialect represent operations on the index type. These include addition and subtraction of (sign-less) indexes. [noa23n]

memref The memref dialect provides operations that target the builtin MemRefType. This includes the declaration of global variables and load and store behaviour. [noa23t]

llvm The llvm dialect defines core LLVM constructs in terms of MLIR. Often this results in one-to-one mappings between MLIR operations and LLVM instructions. Some bridge operations are included that adapt pure MLIR constructs to LLVM. As an example, MLIR uses block arguments instead of ϕ notes; therefore terminator operations can define successor operands, which are forwarded as block arguments at control flow transfer and MLIR does not require an explicit ϕ operation.[noa23s] Another example is differing type systems. Therefore, type conversion is necessary for conversion passes that target the llvm dialect; the infrastructure provides the LLVMTypeConverter.

5 MLIR Brainfuck

MLIR Brainfuck is a provisional but functional Brainfuck compiler based on a MLIR front-end (Bf dialect) and a LLVM back-end (LLVM IR). The compiler is functional since it can be used to execute Brainfuck programs. It is provisional since it is composed of three tools and does not integrate their functionality into one program. The tools are called *bf-to-mlir_bf*, *Bf_opt* and *Bf_translate*. Their usage is described in 5.5, but figure 5.1 provides an overview of the MLIR Brainfuck compiler pipeline that they compose.

The process of transforming Bf to LLVM IR performs the lowering and the translation phase. The lowering phase starts with the Bf dialect and ends with the llvm dialect; it consists of three steps each of them resulting in valid MLIR IR; we call each result a *lowering-product*. Each transformation between lowering products is performed by a conversion pass. The translation phase expects valid MLIR IR composed of llvm dialect constructs and returns LLVM IR.

We introduced the lowering products Optimizable Brainfuck (BfOpt), Explicit Brainfuck (ExplicitBf) and llvm Brainfuck (llvmBf). While OptBf was introduced to enable language-specific optimizations, ExplicitBf makes memory access explicit. It can be used to remove unnecessary memory reads if the memory has not been written since the last read. The final lowering product llvmBf is composed only of constructs of the llvm dialect and is, therefore, input to the Bf-translate tool. To convert between the products, we have the conversion passes --bf-to-optbf (to get from Bf IR to OptBf IR), --optbf-to-explicitbf (to get from OptBf IR to ExplicitBf IR) and --explicitbf-to-llvm (to get from ExplicitBf IR to llvmBf IR). As shown in 5.1 the conversion passes are combined to the pipeline --bf-to-llvm, which is also an Bf-opt option. Figure 5.1 shows that the translation phase is performed by the --mlir-to-llvm-ir option of the Bf-translate tool.

The lowering phase depends on more MLIR dialects than Bf and llvm. A total of 11 dialects are used in the process. We list the dialects in order of their abstraction: Bf, bf_red, builtin, func, scf, arith, index, bf_pointer, memref, cf and llvm.[1] Next to dialects existing in the MLIR ecosystem, we defined and used the custom dialects Bf, bf_red and bf_pointer. In the following, the custom dialects are presented in sections 5.1, 5.2 and 5.3 respectively. Section 5.4 describes the conversion targets and rewrite patterns the conversion passes depend on.

5.1 Bf Dialect

The front end is implemented as an MLIR dialect called Bf. The Bf dialect does not define any types or attributes, since Brainfuck has no type system and the Brainfuck commands can be modelled with operations that do not need custom static data to influence their behaviour.

We defined operations associated to Bf. They can be grouped analogues to the Brainfuck commands in operations with and operations without a region.

[1]Note that these dialects have been briefly described in 4.3.

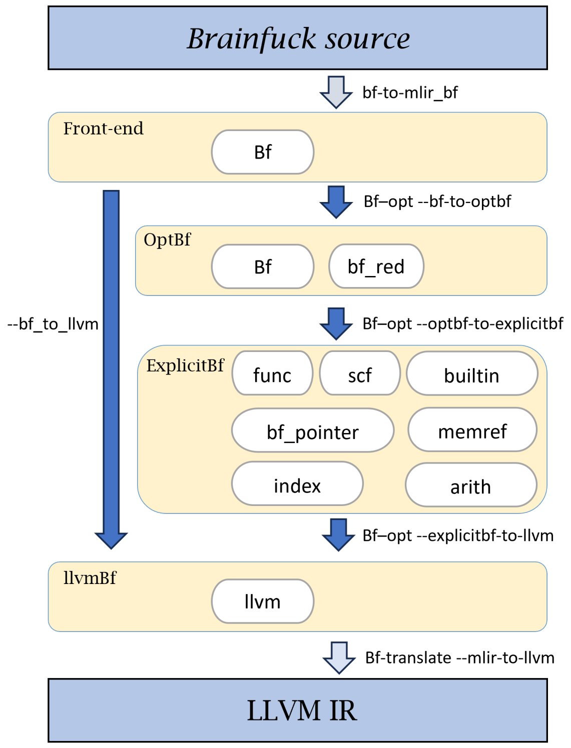

Figure 5.1: MLIR Brainfuck Pipeline. The dark blue arrows are performed by the Bf-opt tool. The upper light blue arrow is performed by the bf-to-mlir_bf tool. The lower light blue arrow is performed by the Bf-translate tool.

5.1.1 Operations without a Region

Each primitive Brainfuck command maps to a MLIR operation without a region. These primitive operations have no region, successor block, SSA operands, attributes or result types; their semantics is only specified by their name and their one-to-one mapping to a Brainfuck command.

The mapping from primitive Brainfuck commands to Bf operations without a region is given in the following listing.

```
                    +            =>            Bf.increment
                    -            =>            Bf.decrement
                    >            =>            Bf.shift_right
                    <            =>            Bf.shift_left
                    .            =>            Bf.output
                    ,            =>            Bf.input
```

Listing 5.1: Mapping from primitive Brainfuck commands to region less Bf operations.

We reduced the custom syntax of the operations to `Bf.<opcode>` since it is the only data subscribing semantic to the operation.

5.1.2 Operations with a Region

A Brainfuck loop consists of the two Brainfuck commands '`[`' and '`]`' and the Brainfuck commands wrapped by them. The Bf dialect models a Brainfuck loop as *one* operation with an associated *constrained* region called body.

The region is constrained to only contain one block. This reflects that Brainfuck has no branching concept. Since a block can contain any (finite) number of operations and an operation can have a region attached, the recursive nature of a Brainfuck loop is captured.

The region is furthermore constrained to be a SSACFG region. This reflects that the Brainfuck commands '+', '-', '.', '<', '>' affect the behaviour of subsequent commands, for they change the values of the Brainfuck pointer or memory. As a consequence, the order of the Brainfuck commands that create a Brainfuck program can, a priori, not be changed without affecting the result of the program: control flow matters in Brainfuck.

A Brainfuck program is defined by Brainfuck code contained in a file ending with '.bf'. One such file thus implicitly forms a Brainfuck module. The Bf dialect supports Brainfuck modules explicitly via an operation (Bf.module) that has an associated region called body. Analogous to and for the same reasons as the body of Bf.loop, it is SSACFG region and can only contain one block.

The mapping from composed Brainfuck commands to Bf operations without a region is given in the following listing; the .. notation refers to Brainfuck commands (Bf operations) contained in a Brainfuck loop or module (region attached to a Bf operation with region).

```
                    [ .. ]          =>            Bf.loop { .. }
                    ( .. , .bf)     =>            Bf.module { .. }
```

Listing 5.2: Mapping from composed Brainfuck commands to Bf operations with region.

The custom syntax of both operations is reduced to the dialect name, opcode idiom and the curly braces that indicate the attached region. The bf-to-mlir_bf tool maps '`[`' to `Bf.loop` { and '`]`' to }; the Bf-opt tool parses the result of bf-to-mlir_bf and detects that a program is not syntactically correct if the Bf dialect loop does not open or close.

5.1.3 Example

The following listing contains a mapping from the example 2.4 to an Bf representation of it. In comparison to the LLVM representation of the feedback loop in 3.4.1, this implementation leaves memory, pointer and console implicit.

```
                                                Bf.module {
                    +               =>              Bf.increment
                    [               =>              Bf.loop {
                        >           =>                  Bf.shift_right
                        ,           =>                  Bf.input
                        .           =>                  Bf.output
                        <           =>                  Bf.shift_left
                    ]               =>              }
                                                }
```

Listing 5.3: Example 2.4 translated to Bf IR.

5.2 bf_red dialect

Analogous to the Bf dialect, bf_red does not define any types or attributes. It has two associated region-less operations.

5.2.1 Operations

The bf_red.increment operation is an abstraction of Bf.increment and Bf.decrement. It expects a signed eight-bit integer attribute as an argument to model the value to be added to the current cell. The bf_red.shift operations is an abstraction Bf.shift_right and Bf.shift_left. It expects a signed 32-bit integer attribute as an argument to model the amount by which the Brainfuck pointer should be shifted. The syntax of the bf_red operations is reduced to the dialect name, and opcode idiom followed by the attribute dictionary in generic textual form.

```
// add amount one to the current cell
bf_red.increment {amount = 1 : si8}

// add amount minus one to the current cell
bf_red.increment {amount = -1 : si8}

// shift the Brainfuck pointer by one
bf_red.shift {value = 1 : si32}

// shift the Brainfuck pointer by minus one
bf_red.shift {value = -1 : si32}
```

Listing 5.4: bf_red Operations.

Note that the bf_red.increment (bf_red.shift) operation can be used to model the behaviour of the Bf.increment and the Bf.decrement operation (Bf.shift_right and the Bf.shift_left operation) by passing the appropriate values as arguments as shown in the listing.

5.3 bf_pointer dialect

The bf_pointer dialect provides an implementation of the Brainfuck pointer. The pointer is a globally accessible structure that stores the write and read position of the program. This implies that the MLIR structure that models it, needs to be referenced in more than just one place. It would be tedious to represent this behaviour using SSA variables. Each write operation would need to store the new value in an unique SSA variable. This would require logic to track the SSA variable that stores the pointers' current value for successive read or write operations to access it. Therefore, the implementation uses MLIR symbols.

In the lowering phase, the dialect will be applied to model memory access in tandem with the memref dialect. Since the memref operations, that are used to access memory, rely on values of type index [noa23t], the bf_pointer dialect serves as a 'wrapper' of the index type.

5.3.1 Operations

To declare the Brainfuck pointer, we define the bf_pointer.ptr operation. It takes a SymbolRefAttr called name and an optional index attribute called inital_value with the default value zero as attributes. Furthermore, it implements the SymbolOpInterface interface. This adds the symbol reference name to the symbol table of the enclosing builtin.module operation. Now, the value of the Brainfuck pointer can be referenced by operations using the name symbol.[sym23, noa21]

To write to and read from the pointer, we need to access the current value of the pointer using the symbol reference name.

We define one operation per functionality. Both implement the SymbolUserOpInterface to enable symbol verification and access. As a result, both operations must define verifying behaviour, which is run before the symbol is accessed.

The bf_pointer.write_ptr operation writes a new value to the pointer. It expects a SymbolRefAttr (called name) and the new pointer value of type index as arguments. The bf_pointer.read_ptr operation reads the current value of the pointer. It takes a SymbolRefAttr (called name) as an argument and returns the current value of type index.

```
// declare the Brainfuck pointer symbol @bf_ptr with initial value zero
of type index.
bf_pointer.ptr @bf_ptr = 0 : (index) -> ()

// read the current value of the Brainfuck pointer
%2 = bf_pointer.read_ptr {name = @bf_ptr} : () -> (index)

// write a new value to the Brainfuck pointer
bf_pointer.write_ptr %2 {name = @bf_ptr}: (index) -> ()
```

Listing 5.5: bf_pointer Operations.

We implemented the verifying behaviour for both operations by checking whether a mapping of the form name $\mapsto$ bf_pointer.ptr exists in the symbol table of the enclosing builtin.module operation.

The syntax of the read and write pointer operations is reduced to contain the dialect name, opcode idiom, the attribute dictionary containing the symbol name and the function type indicating input and output behaviour.

5.4 Lowerings

We present the three lowering-products OpfBf, ExplicitBf and llvmBf respectively. Each lowering product is the result of a conversion pass. The passes are defined by conversion targets composed of legal and illegal operations and rewrite patterns. Each illegal operation requires the definition of a rewrite pattern.

5.4.1 Optimizable Brainfuck

OptBf is composed of operations of the Bf and bf_red dialects. The front-end Bf is lowered to OptBf by the conversion pass --bf-to-optbf. The pass is defined by the conversion target and the rewrite patterns shown in 5.1; if a row contains an illegal op, then also a rewrite pattern must be specified.

Rationale

The operations Bf.increment, Bf.decrement, Bf.shift_right and Bf.shift_left of the Bf dialect can be folded. In the context of the Bf $\rightarrow$ *OptBf* lowering, we call them the *fold-able* operations. The fold-able operations implicitly increment or decrement the current cell (Brainfuck pointer) by one. This can be abstracted to an operation that expects the *signed* value to be added to the current cell (Brainfuck pointer) as an argument. Then only one operation is needed for both increment and decrement (or shift right, shift left). Furthermore, this allows for optimization passes that fold n successive occurrences of the same Bf operation

Illegal Ops	**Legal Ops**	**Rewrite Pattern**
Bf.increment	bf_red.increment	BfIncrementToBfRedLowering
Bf.decrement	bf_red.increment	BfDecrementToBfRedLowering
Bf.shift_right	bf_red.shift	BfShiftRightToShiftLowering
Bf.shift_left	bf_red.shift	BfShiftLeftToShiftLowering
-	Bf.module	-
-	Bf.loop	-
-	Bf.input	-
-	Bf.output	-

Table 5.1: Definition of the Bf $\rightarrow$ $OptBf$ lowering.

to an occurrence of the associated bf_red operation with argument n (or $-n$ if the semantics of the Bf operation involves decreasing).

--bf-to-optbf

The Bf $\rightarrow$ $OptBf$ lowering consists of the four conversion patterns. Each pattern translates one of the fold-able operations to a bf_red operation.

The task of the BfIncrementToBfRedLowering pattern is to translate the Bf.increment operation to the bf_red.increment operation with the value 1 wrapped in an eight-bit signed integer attribute as argument. Analogous, the Bf.decrement operation is translated to bf_red.decrement, but with the value -1 wrapped in an eight-bit signed integer attribute as argument, which is implemented by the BfDecrementToBfRedLowering pattern.

The BfShiftRightToShiftLowering translates the Bf.shift_right operation to the bf_red.shift operation with the value 1 wrapped in a 32-bit signed integer attribute as argument. We need a 32-bit integer, since we assumed our memory size to be 30000 cells which requires a storage size $\geq$ 30000. The Bf.shift_left operation is translated to bf_red.shift, but with the value -1 wrapped in a 32-bit signed integer attribute as argument by the BfShiftRightToShiftLowering conversion pattern. Each pattern wraps the incrementing/shifting value in an attribute since it is known at compile time.

5.4.2 Explicit Brainfuck

The second lowering product is ExplicitBf. It is composed of the dialects bf_pointer, arith, index, func, scf, builtin, memref and llvm. OptBf is lowered to ExplicitBf by the conversion pass --optbf-to-explicitbf. Table 5.2 shows the definition of the pass.

Rationale

Since the core abstraction of the Brainfuck language is the invisibility of memory, pointer and console it implicitly requires read and write operations on the (global) structures memory, pointer and console. For example, the operation bf_red.shift (bf_red.increment) (Bf.input)

Illegal Ops	Legal Ops	Rewrite Pattern
Bf.module	builtin.module, func.func, func.return, memref.global, memref.get_global, memref.dealloc, bf_pointer.ptr	BfModuleLowering
Bf.loop	scf.while, scf.condition, scf.yield memref.get_global, memref.load, bf_pointer.read_ptr, arith.constant, arith.cmpi	LoopOpLowering
bf_red.increment	memref.get_global, bf_pointer.read_ptr, arith.contant, arith.addi, memref.store	BfRedIncrementLowering
bf_red.shift	bf_pointer.read_ptr, index.contant, index.add, bf_pointer.write_ptr	BfRedShiftLowering
Bf.input	func.call, llvm.trunc, memref.get_global, bf_pointer.read_ptr, memref.store	BfInputLowering
Bf.output	func.call, llvm.sext, memref.get_global, bf_pointer.read_ptr, memref.load	BfOutputLowering

Table 5.2: Definition of the $OptBf \rightarrow ExplicitBf$ lowering.

depend on the pointer (pointer and memory) (pointer, memory and console). In ExplicitBf memory, pointer and console are represented with adequate MLIR constructs.

The Brainfuck memory needs to be referenced in more than one place. As with the pointer, it would be tedious to represent this using SSA values. Therefore, to represent the Brainfuck memory and operations on it, we use the memref dialect. It contains operations to declare a symbol for a memory reference and to read from and write to it using the declared symbol [noa23t].

We utilize the custom bf_pointer dialect to declare the Brainfuck pointer and read and write from it.

The console is modelled by the application of the C language functions getchar and putchar. We use the func dialect to declare and call them. Since we lower to LLVM IR and LLVM can call C functions once they are declared in an LLVM module [noa23b], we do not have to define custom MLIR operations for their behaviour.

The implementation of read-and-write access to the Brainfuck memory follows a common structure; see listing 5.4.2.

First, we retrieve the current values of Brainfuck memory and pointer in SSA form (lines one to two). These can be used by successive operations. This part is mandatory for any usage of the memory.

The second part consists of two options A and B. Option A expresses read and option B expresses write access. All rewrite patterns either use option A or option B exclusively or utilize both constructs.

In line three, the current cell is accessed using the memref.load operation on the SSA values %0 and %1. This expresses option A of the second part. Line four expresses option B of the second part. A value is stored to the current cell using the memref.store operation on the memory reference, the index and the value stored in variable %2.

In the example, we restore the value read from the memory. If both options are used in

```
// First part: access the current values of Brainfuck memory and pointer.
%0 = memref.get_global @bf_memory : memref<30000xi8>
%1 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index

// Second part: some lowerings use A some B and some both
// A (optionally): read the current cell.
%2 = memref.load %0[%1] : memref<30000xi8>

// B (optionally): write a new value to the memory. Here we just reuse
the read value.
memref.store %2, %0[%1] : memref<30000xi8>
```

Listing 5.6: ExplicitBf Memory Access Idiom.

a lowering of operation O, the value to be stored in the current cell is calculated from the value of the current cell and operations related to the semantic of O.

BfModuleLowering

The conversion of the Bf.module operation is represented by the conversion pattern BfModuleLowering. Its tasks are to open a builtin.module, declare the symbols related to the Brainfuck memory, pointer and console, define a SSACFG region to be populated with the operations of Bf.module's body region and clean the global structure that models the memory. The following code listing shows the resulting MLIR IR.

```
module {
  bf_pointer.ptr @bf_ptr = 0 : () -> ()
  memref.global "private" @bf_memory : memref<30000xi8>
  func.func private @getchar() -> i32
  func.func private @putchar(i32) -> i32
  func.func @main() {

    // content of Bf.module body region.

    %0 = memref.get_global @bf_memory : memref<30000xi8>
    memref.dealloc %0 : memref<30000xi8>
    return
  }
}
```

Listing 5.7: bf.module → ExplicitBf.

The Brainfuck pointer is declared using the bf_pointer.ptr operation with the initial value zero. We implement the Brainfuck memory in terms of a memory reference of rank 30000 and a sign-less 8-bit integer shape. The operation memref.global adds the symbol @bf_memory to the symbol table of module since builtin.module implements the Symbol interface. It furthermore allocates the requested memory space statically.[noa23t]

The C language functions getchar and the putchar are declared by passing their names as symbols to the func.func operation. Since it implements the Symbol interface, the names of both functions are added to the symbol table of the module [noa23m]. Note that getchar expects and putchar returns a sign-less 32-bit integer. This will lead to casting operations in the conversion of Bf.input and Bf.output.

In line six, the func.func operation is used to declare and define the function that drives the execution of the Brainfuck program. Since in the last lowering step to the llvm dialect, the occurrence `func.func @main` will be transformed to an LLVM function, we declare the function with the `@main` symbol attached, because a LLVM module must contain a main function. We populate the region of func.func operation with the contents of the Bf.module's body region.

In lines 10 to 11 the allocated memory is cleaned.

LoopOpLowering

The conversion of the Bf.loop operation to ExplicitBf is represented by the conversion pattern LoopOpLowering. Its task is to define a scf.while operation and populate its *before* region with the Brainfuck loop condition and its *after* region with the operations contained in the body region of Bf.loop. The following code listing shows the resulting MLIR IR.

```
1     scf.while : () -> () {
2         %0 = memref.get_global @bf_memory : memref<30000xi8>
3         %1 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
4         %2 = memref.load %0[%1] : memref<30000xi8>
5         %c0_i8 = arith.constant 0 : i8
6         %3 = arith.cmpi ne, %2, %c0_i8 : i8
7         scf.condition(%3)
8     } do {
9
10        // content of Bf.loop body region.
11
12        scf.yield
13    }
```

Listing 5.8: bf.loop → ExplicitBf.

We populate the before region with the Brainfuck loop condition. The Brainfuck loop condition checks if the current cell is unequal to zero. Therefore, the value of the current cell is read using the first part and the A option of the ExplicitBf memory access idiom (lines two to three). Then, the value is compared against zero employing the arith.cmpi operation. We apply this operation with the `ne` predicate as argument. This predicate ensures that the two operands are not equal [noa23d]. The scf.condition operation (line seven) is the terminator of the before region. It expects a signless one-bit integer [2] called condition as the first operand; condition indicates whether to execute the after region or to terminate the loop. [noa23y]

The after region is populated with the content of the Bf.loop body region. As terminator the scf.yield operation is applied; this terminates the after region and transfers control flow to the before region [noa23y].

This construction expresses that the Brainfuck loop is essentially a while, not a do-while loop: the loop body is executed only if the Brainfuck loop condition is true.

The recursive application of the LoopOpLowering rewrite pattern, in cases, where the body of a Brainfuck loop again contains a loop, requires implementation of the initialize hook to set setHasBoundedRewriteRecursion, which signalizes that recursion is safely handled [pat23].

[2] i.o.w. a Boolean value, but MLIR does define a Boolean type explicitly.

BfRedIncrementLowering

The conversion of the Bf_red.increment operation to ExplicitBf is represented by the conversion pattern BfRedIncrementLowering. Its tasks are to

1. read the current values of the Brainfuck pointer and memory,
2. read the current cell using the values of Bainfuck pointer and memory,
3. calculate the incremented value,
4. store the incremented value using the values of the Bainfuck pointer and memory.

The following code listing shows the target operation and the resulting MLIR IR.

```
1   // Conversion listing for pass --bf_red-to-arith-bf_pointer
2
3   // Operation to be lowered
4   bf_red.increment {amount = 1 : si8}
5
6   // Resulting MLIR IR
7   %0 = memref.get_global @bf_memory : memref<30000xi8>
8   %1 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
9   %2 = memref.load %0[%1] : memref<30000xi8>
10  %c1_i8 = arith.constant 1 : i8
11  %3 = arith.addi %2, %c1_i8 : i8
12  memref.store %3, %0[%1] : memref<30000xi8>
```

Listing 5.9: bf_red.increment $\rightarrow$ ExplicitBf.

The current cell is read using the first part and option A of the ExplicitBf memory access idiom (lines seven to nine). The amount to be added to the cell is created by applying the arith.constant operation to the amount attribute of bf_red.increment (line 10). Then, the incremented value is calculated with arith.addi used on the value of the current cell and %c1_i8. Finally, option B of the ExplicitBf memory access idiom is applied to %3 to accomplish task four.

BfRedShiftLowering

The conversion of the Bf_red.shift operation to ExplicitBf is represented by the conversion pattern BfRedShiftLowering. Its task is to

1. read the current value of the Brainfuck pointer,
2. calculate the shifted value using the current value and Bf_red.shift.value,
3. store the shifted value.

The following code listing shows the target operation and the resulting MLIR IR.

```
// Conversion listing for pass --bf_red_shift-to-index-and-bf_pointer

// Operation to be lowered
bf_red.shift {value = -1 : si8}

// Resulting MLIR IR
%0 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
```

```
8     %idx1 = index.constant 1 // 1 is the absolute amount the attribute value
9     %1 = index.sub %0, %idx1 // or index.add if -value < 0
10    bf_pointer.write_ptr %1 {name = @bf_ptr} : (index) -> ()
```

Listing 5.10: bf_red.shift → ExplicitBf.

The conversion pass depends on read and write operations on the Brainfuck pointer. Therefore, we use bf_pointer.read_ptr and bf_pointer.write_ptr in lines seven and teen. In between, the value to be added to or subtracted from the current value of the pointer stored in %0 is calculated. Since the index.add and index.sub operations treat their operands as sign-less, a value of type index passed to them can not be meaningfully negative [noa23n]. So, we only create an index constant on the absolute amount of the value attribute of bf_red.shift. In line three, we either use index.sub or index.add depending on the sign of value. In the current example, since value < 0 the index.sub operation is used.

BfInputLowering

The conversion of Bf.input is represented by the conversion pattern BfInputLowering. The task of BfInputLowering is to

1. get a value from the console,
2. get the current values of Brainfuck pointer and memory,
3. store the retrieved value using these values in the current cell.

The following code listing shows the resulting MLIR IR of the BfInputLowering.

```
1  %0 = func.call @getchar() : () -> i32
2  %1 = llvm.trunc %0 : i32 to i8
3  %2 = memref.get_global @bf_memory : memref<30000xi8>
4  %3 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
5  memref.store %1, %2[%3] : memref<30000xi8>
```

Listing 5.11: Bf.input → ExplicitBf.

Task one is implemented by applying the func.call operation to the @getchar symbol. The operation implements the SymbolUserOpInterface. Therefore, it can map the symbol to the declaration of getchar. The function call returns a sign-less 32-bit integer that is stored in %0. Since the memory reference that implements the Brainfuck memory has shape i8, we need to cast the input value before we can store it. In line two, this is done by passing the target type i8 and the operand %0 to llvm.trunc. Finally, the casted value is stored in the Brainfuck memory using the ExplicitBf memory access idiom with option B.

BfOutputLowering

The conversion of Bf.output is represented by the conversion pattern BfoutputLowering. The task of BfOutputLowering is to

1. get the current values of Brainfuck pointer and memory,
2. get the current cell using these values,
3. write the current cell to the console.

The following code listing shows the resulting MLIR IR of the BfOutputLowering.

```
%4 = memref.get_global @bf_memory : memref<30000xi8>
%5 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
%6 = memref.load %4[%5] : memref<30000xi8>
%7 = llvm.sext %6 : i8 to i32
%8 = func.call @putchar(%7) : (i32) -> i32
```

Listing 5.12: Bf.output → ExplicitBf.

We read the current cell using the ExplicitBf memory access idiom with option A. The type of the SSA value %6 is i8. Since the putchar function takes a 32-bit integer as argument, we need to cast. The llvm.sext operation is applied to %6 and the i32 type to accomplish this. At last, func.call is used on the @putchar symbol and the casted value to accomplish step three.

5.4.3 llvm Brainfuck

The fourth lowering product is llvmBf. The conversion pass --bf-to-llvm translates ExplicitBf to the constructs of the llvm dialect. Since ExplicitBf mostly consists of project-independent components, we can utilize existing conversion patterns with only llvm operations as legal operations. This is done for the dialects arith, index, func, scf, builtin, and memref. Note that the lowering of scf to llvm requires conversion to the cf dialect.

As a consequence, we only had to write custom rewrite patterns for the bf_pointer operations. See table 5.3 for an overview of the defined patterns.

Illegal Ops	Legal Ops	Rewrite Pattern
bf_pointer.ptr	llvm.mlir.global	PtrLowering
bf_pointer.read_ptr	llvm.mlir.adressof, llvm.load	ReadPtrLowering
bf_pointer.write_ptr	llvm.mlir.adressof, llvm.store	WritePtrLowering

Table 5.3: Definition of the *Explicit* → *llvmBf* lowering.

Rationale

This conversion expresses the bf_pointer operations in llvm constructs. We utilize the llvm.mlir.global and the llvm.mlir.addressof operation. Both operations contain an additional .mlir prefix, to signalize that the operations relate to concepts not known by the LLVM IR. This results in the additional transformation from MLIR logic to LLVM IR representation, which is implemented in the --mlir-to-llvmir option of the Bf-translate tool.

Since the llvm dialect does not have access to the index type, type conversion is needed. The LLVMTypeConverter is already part of the conversion framework. It provides a function to map the index type to a 64-bit integer; this integer is part of the builtin dialect and is compatible with LLVM's integer type.

PtrLowering

The PtrLowering pattern translates the bf_pointer.ptr operation directly to the llvm.mlir.global operation.

```
   llvm.mlir.global private @bf_ptr(0 : index) {addr_space = 0 : i32} : i64
```

Listing 5.13: bf_red.ptr → llvmBf.

Since the Brainfuck pointer should not be visible outside of the current module, we specify the symbol as private. The value operand of llvm.mlir.global accepts the value 0 wrapped in an index attribute. However note, that the operation returns the i64 type; this is the result of the applied type conversion.

ReadPtrLowering

The task ReadPtrLowering pattern is to get the address of the Brainfuck pointer using the `@bf_ptr` symbol and load the current value of the pointer.

```
%0 = llvm.mlir.addressof @bf_ptr : !llvm.ptr<i64>
%1 = llvm.load %0 : !llvm.ptr<i64>
```

Listing 5.14: bf_red.read_ptr → llvmBf.

The llvm.mlir.adressof operation returns a typed LLVM pointer as required for LLVM memory access. In this case, the signless 64-bit integer type is used, which reflects the type conversion of the index type. The llvm.load Op is applied to the returned address and returns the current value of the pointer as a sign-less 64-bit integer. The custom syntax leaves the return type implicit since it can be deduced from the type $!\mathsf{llvm.ptr} < \mathsf{i64} >$ of the operand.

WritePtrLowering

The task WritePtrLowering pattern is to get the address of the Brainfuck pointer using the `@bf_ptr` symbol and store the new value to the pointer.

```
// This is the converted form of the index.constant 1 operation,
// which was synthesized in the ExplicitBf conversion
%2 = llvm.mlir.constant(1 : i64) : i64


%1 = llvm.mlir.addressof @bf_ptr : !llvm.ptr<i64>
llvm.store %2, %1 : !llvm.ptr<i64>
```

Listing 5.15: bf_red.write_ptr → llvmBf.

5.5 Project Structure, Tools and Usage

MLIR Brainfuck[3] is implemented as a CMake based *standalone*, *out-of-tree* project. In this context, out-of-tree means that the project is not included in the LLVM monorepo. The *CMakeLists.txt* root file verifies existing MLIR and LLVM installations on the computer and uses the path to the installations during the build process.

[3] https://github.com/jacobschw/MLIR-Brainfuck

The project is standalone since it contains an *opt*-like tool to operate on the IR and does not depend on *mlir-opt*: the modular optimization driver, which is contained in a MLIR installation. This tool is intended to test passes: it can parse arbitrary MLIR IR following the rules of the dialects registered in its context[4] and apply registered passes on it.

Project Strucuture The project structure is similar to the structure of the *mlir* subfolder in the LLVM monorepo: header files are stored in the *include* folder and the related implementations/definitions are stored in the *lib* folder; the substructure of both folders is equal. In our context, the main components of the project, which are dialects and conversions, form the substructure.

The custom dialects and passes are implemented using the TableGen framework of the MLIR infrastructure [noa23w, noa23x]. The dialects and passes are declared and partially defined in an associated *.td* file. Since code generation is controlled by CMake constructs provided by MLIR, CMake targets for code generation have to be declared. The code generation targets generate the needed *.h.inc* and *.cpp.inc* files from user-defined *.td* files. The generated files are stored in the *build* folder of the project, which resembles the project structure. They include class declarations, printer and parser logic, getter methods and standard verification logic. Components of the constructs that are not generated (f.e. the concrete implementation of a conversion pass or specific verification code) must be added to the associated *.h*, *.cpp* files.

Additionally, three tools are included in the project. These are the Cpp-based tools Bf-opt, Bf-translate and the python-based tool bf-to-mlir_bf, which translates Brainfuck code to Bf dialect-based front-end.

Bf-opt The tool Bf-opt expects valid MLIR IR and (optional) options as input and returns MLIR IR that results from applying the options to the input IR; it is implemented in close analogy to the mlir-opt tool, but has only access to the dialects used by the project. We use Bf-opt to parse MLIR Brainfuck IR (of arbitrary conversion stages) and apply conversion passes to it.

Bf-translate The tool Bf-translate operates at the boundaries of MLIR; it translates different IR to MLIR or MLIR to different IR [noa22]; with "different IR", we mean IR distinct from MLIR IR. We use the tool to translate llvmBf to LLVM IR by applying the --mlir-to-llvmir option.

Usage To execute a Brainfuck program one has to

1. apply bf-to-mlir_bf to the program,
2. execute Bf-opt --bf-to-llvm on the result of 1,
3. use Bf-translate --mlir-to-llvmir on the result of 2,
4. generate an executable from the LLVM IR using an appropriate compiler (f.e. clang).

The Bf-opt tool can be used on any MLIR IR that is associated with the project (Bf front-end, OptBf, ExplicitBf, llvmBf) by using the appropriate passes; see figure 5.1.

[4]In fact, it can parse any IR constructs that match the generic textual representation, but *unkown* IR constructs can not be verified against their invariants [noa23f].

6 Evaluation

6.1 Results

We present the results of the thesis using the feedback loop example 5.1.3. The following listings shows the result of applying --bf-to-optbf to 5.1.3. OptBf prepares language-specific folding optimizations while leaving memory, pointer and console implicit.

```
Bf.module {                                 Bf.module {
    Bf.increment            =>                  bf_red.increment {amount = 1 : si8}
    Bf.loop {                                   Bf.loop {
        Bf.shift_right      =>                      bf_red.shift {value = 1 : si32}
        Bf.input                                    Bf.input
        Bf.output                                   Bf.output
        Bf.shift_left       =>                      bf_red.shift {value = -1 : si32}
    }                                            }
}                                           }
```

Listing 6.1: Example 2.4 translated to OptBf.

Listing 6.1 gives an implementation of the feedback loop example 2.4 in ExplicitBf IR. It is the result of applying the --optbf-to-explicitbf pass to 6.1. Memory, pointer and console are explicit as they would be in a C implementation, see 2.3.4, and are in the LLVM-based implementation in listing 3.4.1. But the ExplicitBf representation preserves the higher-level control flow of the Brainfuck loop thanks to the scf dialect.

6.2 Dialect and Pass Design

We defend our design choices regarding the implemented dialects and passes.

We wanted to implement a Brainfuck compiler that performed steps one to seven of the classical compiler pipeline based on only one compiler framework. This required the implementation of a dialect that models Brainfuck at the highest possible level. The tool bf-to-mlir_bf confirms this property since it just maps each Brainfuck command to the resulting Bf representation.

The bf_red dialect is required to enable folding optimizations and to simplify conversion; only one conversion pattern is needed for bf_red.shift (bf_red.increment) instead of two for bf.increment and bf.decrement (bf.shift_right and bf.shift_left). The folding optimization is not possible in the context of Bf, since the fold-able operations do not specify any arguments.

The bf_pointer dialect enables the implementation of the Brainfuck pointer as a global variable. This is done by a combination of symbol declaration, symbol-to-SSA materialization [sym23] and symbol write access: the bf_pointer.ptr operation declares the pointer, the bf_pointer.read_ptr turns the current value of the pointer into an index typed SSA value that can be used by successive operations and the bf_pointer.write_ptr operation modularizes the

```
module {
  bf_pointer.ptr @bf_ptr = 0 : () -> ()
  memref.global "private" @bf_memory : memref<30000xi8>
  func.func private @getchar() -> i32
  func.func private @putchar(i32) -> i32
  func.func @main() {
    %0 = call @getchar() : () -> i32
    %1 = llvm.trunc %0 : i32 to i8
    %2 = memref.get_global @bf_memory : memref<30000xi8>
    %3 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
    memref.store %1, %2[%3] : memref<30000xi8>
    scf.while : () -> () {
      %5 = memref.get_global @bf_memory : memref<30000xi8>
      %6 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
      %7 = memref.load %5[%6] : memref<30000xi8>
      %c0_i8 = arith.constant 0 : i8
      %8 = arith.cmpi ne, %7, %c0_i8 : i8
      scf.condition(%8)
    } do {
      %5 = memref.get_global @bf_memory : memref<30000xi8>
      %6 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
      %7 = memref.load %5[%6] : memref<30000xi8>
      %8 = llvm.sext %7 : i8 to i32
      %9 = func.call @putchar(%8) : (i32) -> i32
      %10 = func.call @getchar() : () -> i32
      %11 = llvm.trunc %10 : i32 to i8
      %12 = memref.get_global @bf_memory : memref<30000xi8>
      %13 = bf_pointer.read_ptr {name = @bf_ptr} : () -> index
      memref.store %11, %12[%13] : memref<30000xi8>
      scf.yield
    }
    %4 = memref.get_global @bf_memory : memref<30000xi8>
    memref.dealloc %4 : memref<30000xi8>
    return
  }
}
```

Listing 6.2: Example 2.4 translated to ExplicitBf.

write-access to the pointer. The memref dialect implements a similar approach to model global memory references. As we have seen in the llvmBf conversion section, the *bf_pointer* operations are implemented using constructs of the llvm dialect. We could have used this implementation already in the ExplicitBf representation but choose against it, because we wanted ExplicitBf to be as clean from llvm constructs as possible.

In accordance with the three abstraction levels, we implemented three conversion passes. Instead of writing a fourth pass to enable transforming the front-end directly to the llvm dialect, we implemented a pipeline called --bf-to-llvm that performs the conversion passes successively and is an option of Bf-opt.

Since the project is implemented in MLIR and we took optimization open design choices, MLIR Brainfuck can be easily supplemented by optimization passes on all three levels (OptBf, ExplicitBf, llvmBf). These passes can be integrated to the --bf-to-llvm pipeline at the boundaries of the conversion passes.

6.3 Semantic and Regressions Tests

We added regression tests to the project to gain control over the correctness of the compilation pipeline in the context of possible future changes. The tests depend on a semantic testing approach.

6.3.1 Regression Tests with Lit and Filecheck

We added regression tests using lit [noa23q] and FileCheck [noa23l]. While lit is a LLVM integrated test runner, FileCheck lyou check the outputs of some string-based transformation against a specified expected textual form.

```
// RUN: Bf-opt --<some-option> | FileCheck %s
// CHECK: Check some string
```

Listing 6.3: Example lit/FileCheck Test File.

The tool lit searches for test files and executes the command that follows `// RUN:`. In our case, we use it to execute Bf-opt with some option. The string-based result is passed to FileCheck, which compares it against the expected textual form. The combination of lit and Filecheck helps to verify that the framework recognizes custom constructs (dialects, operations, pass names) and performs passes as expected.

The expected textual form resembles the idea behind the implementation of a pass. If the idea is fraud or the implementation contains an error, so does the expected form. The consequence is that a pass has to be tested against runtime behaviour. We assume the correctness of Bf-translate's --mlir – to – llvmir option. The results of the three conversion steps have to be validated; our approach to semantic testing is given in 6.3.2

After successful end-to-end testing, we assumed the correctness of each conversion pass. From this point on, the test suite truly serves for regression test since malicious changes in the code base can be easily tracked.

6.3.2 Semantic Test

The approach is to run the MLIR Brainfuck compiler on simple programs and execute the result using the clang compiler. We exploit that Brainfuck supports console output. Our first step is to verify the Brainfuck program '`,.`': we require the output for tests of the other Brainfuck commands and we need something different from the null bit to ensure that the output is working. After ensuring that the output works, we can test increment, decrement, shift left and shift right commands using appropriate programs. An example is the following program in Bf notation to test the increment command.

```
Bf.module {
    Bf.output
    Bf.increment
    Bf.output
}
```

Listing 6.4: Semantic Increment Test in Bf Notation.

The first output command does nothing because the cells are initialized with zero. We expect the second output to result in the sign related to the ASCII code 1: a transparent smiling smiley.

Using the correctness of the increment command allows us to test the loop commands and the cell wrapping mechanism.

```
Bf.module {
    Bf.increment
    Bf.output

    Bf.loop {
        Bf.increment
        Bf.output
    }

    Bf.increment
    Bf.output
}
```

Listing 6.5: Semantic Loop/Cell Wrap Test in Bf Notation.

Since we verified the increment command, we expect the first output to result in a transparent smiling smiley. Accordingly, the current cell is unequal to zero and the loop should execute until the current cell equals zero. Since the loop body first increments and then outputs the result, we expect every sign related to an ASCII code to be printed. Then the cell should wrap, or in other words, the current cell should become zero. Therefore, the third output should show the transparent smiling smiley again.

Tests for the shifting commands are yet missing. Since the Brainfuck pointer is initialized with zero and MLIR Brainfuck does not support tape wrapping, testing the shift left command depends on the correctness of the shift right command.

The shift right command is tested using the results of the output and the input operation.

```
Bf.module {
    Bf.increment
    Bf.output
    Bf.shift_right
    Bf.increment
    Bf.output
}
```

Listing 6.6: Semantic Shift Right Test in Bf Notation.

As always, we expect the first output to show a transparent smiling smiley. Then, we apply the shift right command, so the current cell $cell_i$ should fulfil $cell_i \neq cell_0$. This is true if the second output results in a transparent smiling smiley: since $cell_i$ was initialized to zero, any different result would be an error.

Currently, we only verified that the shift operation moves the pointer to some cell different from $cell_0$. We use this knowledge to test the shift left command.

```
Bf.module {
    Bf.shift_right
    Bf.increment
    Bf.shift_right
    Bf.increment

    Bf.loop {
        Bf.output
        Bf.shift_left
    }
}
```

Listing 6.7: Semantic Shift Left Test in Bf Notation.

The program starts with shifting the pointer to the right. The $cell_i$ is marked using an increment command. Therefore, the following loop executes and we expect the first output to be the transparent smiling smiley. We expect the following shift left command to move the pointer to $cell_0$. Since $cell_0$ was initialized to zero and stayed untouched since then, it should be zero and the loop terminates.

After we gained confidence that the single Brainfuck commands were converted correctly, we tested the MLIR Brainfuck pipeline with larger Brainfuck programs. The programs, we tested can be found in the *bf_scripts* subfolder of our git repository. Starting with a hello world program (hello_world.bf) [noa23e], we continued with a list of programs competing for the shortest possible hello world Brainfuck program (hw_i.bf, $i \in 1, .., 4$) [noa23g]; these programs use rather advanced techniques and require loops to be executed often. We included a non-terminating factorials calculator (factorials.bf) [26223a]; during the development of our LLVM-based Brainfuck compiler, we found that the storage size of our Brainfuck pointer was not large enough because the program terminated after printing the first ten factorials. We finished the test stack with a Brainfuck program that outputs a mandelbrot set (mandelbrot.bf) [noa23o]. The listed programs form our test stack, which is to be executed after changes are introduced. Additionally, we tested all programs included in bf_scripts.

7 Related Work

We give a brief overview of related work focusing on thematic and methodical similarity as well as prospects and inspirations for future work.

7.1 MLIR ONNX Compiler

Jin et al. [JBL+20] implemented a compiler for ONNX Neural Network Models in MLIR. Open Neural Network Exchange (ONNX) is an open-source framework to assist portability of machine learning models between different environments. The compiler takes a machine learning model specified in ONNX as input and rewrites the model to native code for target hardware. This compiler is implemented using MLIR.[JBL+20]

MLIR Brainfuck is not related thematically to the work of Jin et al. but methodically: their compiler lowers a MLIR dialect used to represent ONNX models down to LLVM IR in three steps. They implemented the onnx dialect to represent ONNX models, the krnl dialect that provides a common lowering point for operations of the onnx dialect and conversion passes from onnx $\rightarrow$ krnl, krnl $\rightarrow$ llvm; the second conversion pass is composed of the passes krnl $\rightarrow$ affine, std and affine, std $\rightarrow$ llvm. Different to MLIR Brainfuck they implemented optimization passes on all three intermediate representations (onnx, krnl and affine, std).[JBL+20]

Their presentation of the project focuses on dialect and optimization pass design, while our presentation of MLIR Brainfuck focuses on dialect and conversion pass design. In particular, Jin et al. describe how to express optimization passes using MLIR's Declarative Rewriting Rules (DRR) [noa23z]. Therefore, both projects taken together could be of methodical use for future MLIR-based projects. A further inspiration we can take from Jin et. al. is their integration of the tasks performed by our three tools into one compiler program.

7.2 Polygeist

Polygeist is a compilation flow that enables C and Cpp compilation using polyhedral MLIR implemented by Moses et. al [MCZZ21]. The project consists of a C/Cpp front-end that translates the clang AST to MLIR's scf dialect to preserve higher-level control flow, a raising of this representation to an affine representation, a bi-directional conversion to and from an exchange format to apply polyhedral tools to the affine based MLIR IR and a back-end to run post-processing MLIR optimizations before converting to an executable.

The affine dialect [noa23c] models constructs of a program in a way that is representable in the polyhedral model. The polyhedral model is based on a linear algebraic representation of programs constructs [Bas04]. An example is for-loops since they have bounds and induction variables expressible by affine-linear mappings.

MLIR Brainfuck is related to Polygeist thematically and methodically: first, as Polygeist, MLIR Brainfuck maps constructs of a programming language to MLIR representation, and second, as Polygeist, MLIR Brainfuck exploits multiple MLIR dialects to lower the initial representation to an executable; in particular, the scf and the memref dialect are used to model control flow and, respectively, memory access. Further note that any Brainfuck program is a possible input to Polygeist since Brainfuck is expressible in C 2.3.4.

MLIR Brainfuck differs from Polygeist in that it does not exploit the MLIR representation to optimize programs. Since we expressed control flow using the scf dialect and every Brainfuck command requires some memory access, we expect we could exploit the affine dialect in a way similar to Polygeist.

8 Discussion

8.1 Limitations

We did not integrate the tools bf-to-mlir_bf, Bf-opt and Bf-translate into one compiler. This is because our focus lies on experimenting with MLIR and the Bf-opt tool provides out-of-the-box possibilities to play around with your custom and the existing IR. Additionally, the compiler pipeline does not support custom Brainfuck memory size or Brainfuck pointer initialization. Finally, we did not investigate applying the DRR framework to conversion passes. It would be interesting to see if our rewrite patterns can be specified declaratively using this framework.

8.2 Opportunities and Future Work

8.2.1 Optimizations

Currently, optimizations are still missing. We neither applied existing MLIR optimization passes, nor implemented custom ones. Since we added the abstraction layers OptBf and ExplicitBf optimization passes on both levels can be integrated. On the OptBf level sequences of bf_red.increment (bf_red.shift) operations can be folded. On the ExplicitBf level, unnecessary memref.get_global operations can be removed. We can use Jin et. al. [JBL+20] as a starting point since they provided optimization passes using the DDR rewriting rules and the Cpp API. To make existing MLIR passes applicable, a starting point could be to add traits (or interfaces) to the definition of the operations of Bf or bf_red. Furthermore, since we lower to LLVM IR, after integrating the project into a compiler program, module-level optimization passes of LLVM can be applied.

8.2.2 Raise MLIR Brainfuck to affine

As described in section 7.2 Moses et. al. implemented a C/Cpp compiler in MLIR. The compiler optimizes C/Cpp code using polyhedral models. Their approach depends on translating specific constructs C/Cpp constructs to the affine dialect.

It is an interesting question what Brainfuck constructs can be translated into affine constructs. Since Brainfuck is Turing-complete, any loop is computable in Brainfuck. In particular, for-loops should be expressible.

The task is to identify the loop bounds and the induction variable of a Brainfuck loop in order to represent it as a affine for-loop. Many Brainfuck programs use a fixed cell to decrement a loop counter down to zero; it could be a start to identify such patterns and represent them as (loop start $\rightarrow$ initial value of the cell, loop end $\rightarrow$ the cell takes on zero, loop step $\rightarrow$ the decreasing amount of the cell per loop execution).

Since every Brainfuck command requires memory access, in the context of a for-loop presentation of a Brainfuck loop also the specific values of the Brainfuck pointer should be in the image of an affine-linear mapping of an induction variable [noa23c]. An approach could be to add an induction variable for each cell that is accessed in the loop body, identify access rules for the cells (possible similarities in the values of the Brainfuck pointer whenever a specific cell is accessed) and represent them as affine-linear mappings of the according induction variable.

The overall approach could be to define affine patterns in Brainfuck, create a dialect for them (AffineBf), lower the Bf dialect (or OpfBf or ExplicitBf or something in between) to AffineBf, lower the AffineBf dialect to affine.

8.3 Takeaways on working with MLIR

We give a brief discussion on takeaways from working with MLIR.

MLIR Standalone The standalone example project in *llvm/mlir/examples* is a good starting point. It contains a *CMakeLists.txt* root file for standalone out-of-tree projects and the structure for the typical project components dialects, types and transformations. Additionally, it contains the tools Bf-opt and Bf-translate to operate on the IR and translate the IR to MLIR extern IR.

The mlir folder structure If you want to introduce components like attributes or conversion passes study the structure of the MLIR subfolder in the LLVM monorepo. The distinction between *Dialect*, *Transforms* and *Conversion* is important. Since the dialects stored in *Dialect* are extensions of the MLIR IR, therefore the structure is comparable to the structure of out-of-tree projects.

The build folder The *build* folder contains an *include* subfolder. This folder contains the CMake targets to generate documentation and code and stores the generated code in *.inc* files. Read the generated files if you are not sure what extra definitions are to be manually added. Another approach is to build the project and determine the missing pieces from the error messages.

Defining Conversion Passes The toy tutorial gives a brief introduction to define conversion passes. For more information the *Conversion* subfolder of the mlir folder contains many examples of existing conversions.

9 Conclusion

We implemented a provisional Brainfuck compiler based on MLIR. Import, MLIR inherent lowering and translation to LLVM IR require using distinct tools. The MLIR inherent lowering starts from a high-level representation of the Brainfuck language, introduces two additional levels of abstraction called OptBf and ExplicitBf and targets the MLIR representation of LLVM. We found that the Brainfuck language despite being minimalist, requires the implementation to use existing dialects next to three custom dialects Bf, bf_red, bf_pointer. The dialects were implemented with the TableGen-based code generation infrastructure ODS [noa23w]. This enabled us to focus on the semantics of the constructs and ignore implementation details. Although we declared the conversion passes with the code generation infrastructure, which implemented the embedding of the new passes in the MLIR pass infrastructure, the core conversion logic was defined using the Cpp API; we did not investigate in using the Declarative Rewrite Rule (DRR) framework [noa23z].

Interesting perspectives for future work are the implementation of language-specific optimizations on our abstraction levels, the formulation of our rewrite patterns in DRR where possible and raising MLIR Brainfuck to the affine dialect.

While MLIR is a heavy framework that requires a deep dive into its concepts and extension architecture, investing in it pays off by enabling users to focus on the special characteristics of their project.

Listings

List of Figures

List of Tables

Bibliography

[26223a] 262588213843476: *Brainfuck interpreter.* https://gist.github.com/mosra/993799. Version: November 2023

[26223b] 262588213843476: *Original brainfuck distribution by Urban Müller.* https://gist.github.com/rdebath/0ca09ec0fdcf3f82478f. Version: November 2023

[App98] APPEL, Andrew W.: SSA is functional programming. In: *ACM SIGPLAN Notices* 33 (1998), April, Nr. 4, 17–20. http://dx.doi.org/10.1145/278283.278285. – DOI 10.1145/278283.278285. – ISSN 0362–1340, 1558–1160

[Bas04] BASTOUL, Cédric: Code Generation in the Polyhedral Model Is Easier Than You Think, IEEE Computer Society, 2004 (29 September - 3 October 2004, Antibes Juan-les-Pins, France), 7–16

[CFR+91] CYTRON, Ron ; FERRANTE, Jeanne ; ROSEN, Barry K. ; WEGMAN, Mark N. ; ZADECK, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Oktober, Nr. 4, 451–490. http://dx.doi.org/10.1145/115372.115320. – DOI 10.1145/115372.115320. – ISSN 0164–0925, 1558–4593

[Cri23] CRISTOFANI, Daniel B.: *A universal Turing machine in Brainfuck.* http://brainfuck.org/utm.b. Version: September 2023

[JBL+20] JIN, Tian ; BERCEA, Gheorghe-Teodor ; LE, Tung D. ; CHEN, Tong ; SU, Gong ; IMAI, Haruki ; NEGISHI, Yasushi ; LEU, Anh ; O'BRIEN, Kevin ; KAWACHIYA, Kiyokuni ; EICHENBERGER, Alexandre E.: *Compiling ONNX Neural Network Models Using MLIR.* http://arxiv.org/abs/2008.08272. Version: September 2020. – arXiv:2008.08272 [cs]

[LA04] LATTNER, C. ; ADVE, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* San Jose, CA, USA : IEEE, 2004. – ISBN 978–0–7695–2102–2, 75–86

[LAB+21] LATTNER, Chris ; AMINI, Mehdi ; BONDHUGULA, Uday ; COHEN, Albert ; DAVIS, Andy ; PIENAAR, Jacques ; RIDDLE, River ; SHPEISMAN, Tatiana ; VASILACHE, Nicolas ; ZINENKO, Oleksandr: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* Seoul, Korea (South) : IEEE, Februar 2021. – ISBN 978–1–72818–613–9, 2–14

[MCZZ21] MOSES, William S. ; CHELINI, Lorenzo ; ZHAO, Ruizhe ; ZINENKO, Oleksandr: Polygeist: Raising C to Polyhedral MLIR. In: *2021 30th International Conference* *on Parallel Architectures and Compilation Techniques (PACT)*. Atlanta, GA, USA : IEEE, September 2021. – ISBN 978–1–66544–278–7, 45–59

[MN21] McCaskey, Alexander ; Nguyen, Thien: *A MLIR Dialect for Quantum Assembly Languages*. `http://arxiv.org/abs/2101.11365`. Version: Januar 2021. – arXiv:2101.11365 [quant-ph]

[Mor15] Morr, Sebastian: Esoteric Programming Languages. (2015)

[noa21] *Globals in MLIR - MLIR*. `https://discourse.llvm.org/t/globals-in-mlir/4187`. Version: August 2021. – Section: MLIR

[noa22] *[doc] mlir-translate / mlir-opt - MLIR*. `https://discourse.llvm.org/t/doc-mlir-translate-mlir-opt/60751`. Version: März 2022. – Section: MLIR

[noa23a] *1 TableGen Programmer's Reference — LLVM 18.0.0git documentation*. `https://llvm.org/docs/TableGen/ProgRef.html`. Version: November 2023

[noa23b] *4. Kaleidoscope: Adding JIT and Optimizer Support — LLVM 18.0.0git documentation*. `https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl04.html`. Version: Oktober 2023

[noa23c] *'affine' Dialect - MLIR*. `https://mlir.llvm.org/docs/Dialects/Affine/#polyhedral-structures`. Version: November 2023

[noa23d] *'arith' Dialect - MLIR*. `https://mlir.llvm.org/docs/Dialects/ArithOps/#arithsubi-arithsubiop`. Version: Oktober 2023

[noa23e] *brainfuck - Esolang*. `https://esolangs.org/wiki/Brainfuck`. Version: September 2023

[noa23f] *Chapter 2: Emitting Basic MLIR - MLIR*. `https://mlir.llvm.org/docs/Tutorials/Toy/Ch-2/`. Version: Oktober 2023

[noa23g] *code golf - "Hello, World!" - Code Golf Stack Exchange*. `https://codegolf.stackexchange.com/questions/55422/hello-world/68494#68494`. Version: November 2023

[noa23h] *Computational class - Esolang*. `https://esolangs.org/wiki/Computational_class#Simulation`. Version: November 2023

[noa23i] *Defining Dialects - MLIR*. `https://mlir.llvm.org/docs/DefiningDialects/`. Version: November 2023

[noa23j] *Dialect Conversion - MLIR*. `https://mlir.llvm.org/docs/DialectConversion/#rewrite-pattern-specification`. Version: Oktober 2023

[noa23k] *Dialects - MLIR*. `https://mlir.llvm.org/docs/Dialects/`. Version: November 2023

[noa23l] *FileCheck - Flexible pattern matching file verifier — LLVM 18.0.0git documentation.* https://llvm.org/docs/CommandGuide/FileCheck.html. Version: November 2023

[noa23m] *'func' Dialect - MLIR.* https://mlir.llvm.org/docs/Dialects/Func/. Version: Oktober 2023

[noa23n] *'index' Dialect - MLIR.* https://mlir.llvm.org/docs/Dialects/IndexOps/#indexconstant-indexconstantop. Version: Oktober 2023

[noa23o] *Index of /brainfuck/bf-source/prog.* http://esoteric.sange.fi/brainfuck/bf-source/prog/. Version: November 2023

[noa23p] *Interfaces - MLIR.* https://mlir.llvm.org/docs/Interfaces/#regionkindinterfaces. Version: Oktober 2023

[noa23q] *lit - LLVM Integrated Tester — LLVM 18.0.0git documentation.* https://llvm.org/docs/CommandGuide/lit.html. Version: November 2023

[noa23r] *The LLVM Compiler Infrastructure Project.* https://llvm.org/. Version: November 2023

[noa23s] *'llvm' Dialect - MLIR.* https://mlir.llvm.org/docs/Dialects/LLVM/#llvmload-llvmloadop. Version: Oktober 2023

[noa23t] *'memref' Dialect - MLIR.* https://mlir.llvm.org/docs/Dialects/MemRef/#memrefglobal-memrefglobalop. Version: Oktober 2023

[noa23u] *MLIR Language Reference - MLIR.* https://mlir.llvm.org/docs/LangRef/#builtin-operations. Version: Oktober 2023

[noa23v] *MLIR Rationale - MLIR.* https://mlir.llvm.org/docs/Rationale/Rationale/#integer-signedness-semantics. Version: November 2023

[noa23w] *Operation Definition Specification (ODS) - MLIR.* https://mlir.llvm.org/docs/DefiningDialects/Operations/#declarative-assembly-format. Version: September 2023

[noa23x] *Pass Infrastructure - MLIR.* https://mlir.llvm.org/docs/PassManagement/#pass-manager. Version: November 2023

[noa23y] *'scf' Dialect - MLIR.* https://mlir.llvm.org/docs/Dialects/SCFDialect/#scfyield-scfyieldop. Version: Oktober 2023

[noa23z] *Table-driven Declarative Rewrite Rule (DRR) - MLIR.* https://mlir.llvm.org/docs/DeclarativeRewrites/. Version: November 2023

[pat23] *Pattern Rewriting : Generic DAG-to-DAG Rewriting - MLIR.* https://mlir.llvm.org/docs/PatternRewriter/#initialization. Version: November 2023

[sym23] *Symbols and Symbol Tables - MLIR.* https://mlir.llvm.org/docs/SymbolsAndSymbolTables/. Version: Oktober 2023

[tra23] *Traits - MLIR*. https://mlir.llvm.org/docs/Traits/. Version: November 2023

[ULSA08] ULLMAN, Jeffrey D. ; LAM, Monica S. ; SETHI, Ravi ; AHO, Alfred V.: *Compiler*. https://elibrary-pearson-de.emedien.ub.uni-muenchen.de/book/99.150005/9783863265748. Version: Januar 2008. – ISBN: 9783863265748 Publisher: Pearson Deutschland

[WG84] WAITE, W. M. ; GOOS, Gerhard: *Compiler construction*. New York : Springer-Verlag, 1984 (Texts and monographs in computer science). – ISBN 978–0–387–90821–2