Pinch: A simple borrow checked language built with LLVM MLIR

Austin Shafer
amshafe2@ncsu.edu
North Carolina State University
Raleigh, North Carolina

CCS CONCEPTS

• **Compilers**; • **Static Analysis**;

KEYWORDS

Compilers, Borrow Checking

1 ABSTRACT

LLVM's Multi-Level Intermediate Representation (MLIR) project creates an extensible ecosystem of IR dialects that allows high level languages to share optimizations. One of the advertised abilities of MLIR is borrow checking: validation of all variable access operations at compile time. Borrow checking has been popularized by the Rust language, where it is used to ensure safety and automatically free data without garbage collection. In this paper, we create a simple borrow checked language built on LLVM MLIR, named Pinch. Pinch uses an ownership graph encoded in MLIR operation attributes to perform validation of memory accesses in a lexical scope.

2 INTRODUCTION

Many high level domain specific languages have been built using the LLVM compiler infrastructure. Many of these languages implement their own high level intermediate representation (IR) on top of LLVM's IR. Noticing that these different projects kept re-implementing common functionality for their high level IRs, LLVM created a new Multi-level Intermediate Representation (MLIR).

LLVM MLIR allows for high level languages to share code through the use of MLIR dialects. MLIR "dialects" capture the operations of one language, allowing other projects and reuse optimization and code generation passes. A new dialect can be created for a domain specific language and then lowered through other dialects to the LLVM IR dialect. Once the high level MLIR has been converted to LLVM IR code can be generated. MLIR is extensively used for Tensorflow, GPU kernels, and affine transformations.

The core component of MLIR is the operation. Operations are very flexible, and can encapsulate both the action itself along with any named attributes that have been attached to it. Operation names follow the pattern "DialectName.OperationName". LLVM provides a tutorial for MLIR that makes a simple tensorflow inspired language named Toy.

One of the proposed capabilities of MLIR is the creation of borrow checked languages. Borrow checking is a form of static analysis which prevents unsafe operations and it provides an alternative to runtime memory cleanup solutions such as garbage collection. It verifies at compile time that all pointer operations take place within the valid lifetime of the variable they point to, and automatically frees data when its lifetime ends. The Rust language has risen to new popularity in recent years, showing that borrow checking can be effectively implemented and utilized in practice.

Borrow checking attributes a lifetime to all named variables, each of which "own" some data. Data may only have one owner, and temporary access to owned data can be loaned out in the form of references. Multiple read-only shared references or one mutable (read-write) reference may be "borrowed", so long as the variable they borrow from is still alive. Programmers in a borrow checked language are forced to write programs that are valid when tested against the borrow checker's rules. Any violation of these safety rules results in a compiler error.

Pinch is a simple borrow checked language inspired by LLVM's Toy language. It allows for simple pointer and arithmetic operations, along with functions and basic variable printing. All operations are validated by a borrow checker, implemented as an LLVM Compiler Pass. The Pinch IR is implemented as an MLIR dialect, which is then lowered through the standard dialect into LLVM IR for code generation. All Pinch MLIR operations are tagged with a source and destination to form an dataflow graph representing ownership. This ownership graph is analysed during the borrow checking pass to ensure program safety.

In this paper we present the following contributions:

- Show a possible implementation of a borrow checked language in MLIR.
- Demonstrate how the available MLIR dialects aided this development.

By implementing Pinch we show that MLIR is indeed more than capable of creating borrow checked languages. Although it is not likely that existing projects switch to MLIR overnight, we hope to demonstrate its potential to future developers.

3 THE PINCH LANGUAGE

The pinch language is lexically intended to appear similar to a watered-down version of the Rust language. It supports integer addition and multiplication, function calls, pointers, and has a builtin print function. The only types are a 32-bit unsigned integer, represented by the token 'u32', and a heap allocated box. Lines are semicolon-terminated.

Declaring a variable begins with the token 'let':

```
let a = 3;
```

Types are inferred automatically, as all variables are either a u32 or a reference. In the code above variable 'a' is initialized as a constant, and is therefore a u32 type. We can borrow a shared reference to 'a' to create a pointer:

```
let a = 3;
let b = &a;   // borrow the value owned by a
let c = &a;
let d = *b;   // dereference pointer b
```

As shown above, multiple read-only references can exist at one time. A mutable reference is exclusive however, and does not allow this. Mutable references can be taken with the '&mut ' operator.

Pinch also allows for heap allocation, which allows us to demonstrate lifetimes of variables which do not end with the termination of their lexical scope. The 'Box' type is a pointer to a heap allocated u32.

```
let b = box(3);
```

A 'Box' type is created by calling the 'box' function and passing the value that the Box should be initialized with.

Functions follow the usual calling convention, and can be defined as such:

```
fn function_name(arg: type, ...) -> return_type
```

We can now put it all together for a valid example:

```
fn main() {
    let a = 4;
    let b = &mut a;
    print(*b);
}
```

This is an executable Pinch program, since it has a main function. We create a stack allocated variable 'a', borrow a write-only mutable reference to it and store that reference in variable 'b', then dereference our pointer and print the value. Variable 'a' has a lifetime of this main function's scope. This is safe and will pass borrow checking because the mutable reference exists inside the valid lifetime for the variable it references. Since there is a mutable reference borrowed, 'a' cannot be used.

Now let's take a look at some examples where the Pinch borrow checker will generate errors as a result of unsafe operations. Borrow checking in Pinch follows a few rules:

- Data is owned by one and only one named variable
- Multiple shared references pointing to an owned data can be borrowed
- One mutable reference pointing to an owned data can be borrowed

References essentially model the multiple reader/one writer problem. Shared references are read only pointers and mutable references are exclusive writable pointers. Only one of the latter two rules can be in place at once. If a mutable pointer exists then no shared references may exist.

This can be demonstrated with an example. The following will fail the borrow check:

```
let a = 4;
let b = &mut a;
let c = &a;     // generates an error
```

Variable 'b' is a mutable reference to 'a' which is active for the rest of this lexical scope. As a result, we cannot take another shared reference to 'a', since it is already borrowed mutably. Pinch will prevent this from compiling as its execution could be unsafe if the value was updated through 'b' while being read by 'c'.

Another check performed is ensuring that references do not outlive the variable they point to. The lifetimes of variables are bound to their lexical scope, and references to them should not exceed this:

```
fn invalid_function(arg: &u32) -> &u32 {
    let ret = *arg * 2;
    return &ret;
}
```

In the invalid function above we stack allocate a u32 variable named 'ret'. We then return a reference to 'ret', which is a pointer to its stack address. When the scope ends 'ret' will be deallocated and the reference we return would be a dangling pointer. Pinch detects this, and generates an error that the reference outlives its value.

Although this example is demonstrated with stack allocated data, it generalizes to any data type. If a reference to a heap allocation outlives the duration of that heap allocation the same problem exists.

4 CHALLENGES

The largest challenge by far was understanding the LLVM programming interface and how to work with MLIR. There are some documents online that help the development process, but you need to get up to speed on both MLIR's api and LLVM's utility classes. It was also tricky to find an extensible solution to borrow checking, and trying to investigate what Rust does was not helpful, as their IR is nothing like MLIR. Most of the transformations Rust does with IR lowering involves canonicalization of complex Rust syntax. Once I settled on the idea of an ownership graph, things became much more clear. Lowering to code generation also posed a large problem, mostly because I was not familiar with LLVM. Significant rewrites were needed, and I could not reference the Toy examples.

5 IMPLEMENTATION

Pinch's borrow checking is implemented as an LLVM MLIR compiler pass. An AST is first generated, and Pinch MLIR is generated in its own dialect. Once the MLIR is generated the borrow checker runs, emitting any errors as a result of unsafe operations. After the borrow checking has succeeded, functions can be inlined and the high level Pinch MLIR can be lowered to LLVM IR through the Standard MLIR dialect. Code can then be generated from the LLVM IR as usual.

The lexer and parser for AST construction are a heavily modified version of Toy. Toy is a language for performing tensor calculations, and so large modifications were needed to remove tensors and add references. Pinch only has two base data types: a heap-allocated Box and an unsigned 32-bit integer (ui32). Toy also does some type detection and reshaping automatically, and so extra logic was added to Pinch's parser to check types, such as preventing passing a reference as an integer.

Once a reference-aware AST had been generated we could construct the Pinch IR. Pinch's MLIR dialect contains some common operations, such as constants and function calls, along with some borrow-checking specific operations. The borrow checking operations signify the transfer or reference of data. The move operation moves ownership of a value from one named variable to another. The borrow operations create a reference to their operand. All pinch operations track ownership through attribute dictionaries, which need to be constructed by the parser.

The first pass to run on the Pinch MLIR is the borrow checking pass. This is effectively the portion of the project which is the actual "borrow checker". The borrow checker processes each function in parallel and verifies all operations within that functions lexical scope. For simplicity, the lifetime during which a variable is alive is based on that variable's lexical scope. When that variables lexical scope ends it will be freed.

To explain the borrow checker we first need to examine the attributes that make up the ownership graph. All operations are tagged with attributes specifying the source and destination of values that they use. These attributes hold a string uniquely identifying the name of the owner of that value. As it processes each operation the borrow checker tracks information for each owner, such as the type and number of references to its data. If a name has not been seen before, then a new owner is made from it so it can be identified in later operations.

Below is a an example which moves the owned data from variable b to variable a:

```
let a = 2;
let b = a;
```

And here is the Pinch MLIR generated:

```
%0 = pinch.constant {dst = "a"} 2
%1 = "pinch.move"(%0) {dst = "b", src = "a"}
    : (ui32) -> ui32
```

First a new SSA value is initialized from a constant. The destination of this constant operation is marked as 'a' in the 'dst' attribute. This signifies that we are creating a new owner. With the "pinch.move" operation we are once again creating a new owner based on the 'dst' attribute, but we also specify where the moved data originated from with the 'src' attribute. When the borrow checker is validating this operation it can mark owner 'a' as non-resident, and any further access of 'a' should trigger an error.

After the borrow checker has verified that all operations are safe we are ready to generate LLVM IR. MLIR encapsulates LLVM IR as the LLVM dialect, which other dialects can be lowered to. This was implemented as multiple passes lowering the Pinch MLIR down to code generation. Two lowering passes had to be written for this to work: lowering to the standard dialect, and lowering from the standard dialect to the LLVM dialect. Additional passes were reused from the MLIR infrastructure, such as inlining function calls and removing common subexpressions. LLVM also takes care of generating code from the LLVM MLIR dialect.

Although the lowering process is inspired by Toy, the implementation of its lowering passes differs significantly. Pinch needs to stack allocate variables and create pointers to those allocations whenever a borrow occurs. Loads also need to be generated whenever a variable is used or dereferenced.

6 PROGRAM EXAMPLES

Please note that due to formatting some MLIR operations may span multiple lines. Indentation has been added on wrapped lines in an attempt to make things more readable. Also note that compilation errors shown have been trimmed. In practice, all errors include the source code location that caused them.

First let's look at a valid Pinch program:

```
fn main() {
   let a = 1 + 2;
   let b = &mut a;
   print(*b);
}
```

Here we make a stack variable 'a', borrow a mutable reference to it, and print the data that reference points to. This will generate the following MLIR:

```
module {
  func @main() attributes {src = []} {
    %0 = pinch.constant {dst = ""} 1
    %1 = pinch.constant {dst = ""} 2
    %2 = pinch.add %0, %1 {dst = "a"}
        : ui32
    %3 = "pinch.borrow_mut"(%2)
        {dst = "b", src = "a"}
        : (ui32) -> memref<1xui32>
    %4 = "pinch.deref"(%3)
        {dst = "", src = "b"}
        : (memref<1xui32>) -> ui32
    pinch.print %4 : ui32
    pinch.return {src = ""}
  }
}
```

If executed, this will print the value 3. The first half of this MLIR looks similar to the example Toy language, and the latter portion is much more specific to Pinch. Temporary variables have their 'dst' attribute set to an empty string to signify that they are temporary and not owned by a named variable in the original source code. This code is safe and will pass borrow checking.

Since we are discussing borrow checking, let's examine an example where the borrow checker generates an error:

```
let a = 42;
let b = &mut a;
let c = &a;     // generates an error
```

This is our example from earlier, where we incorrectly tried to take a shared reference after a mutable reference. The following MLIR is generated:

```
module {
  func @main() attributes {src = []} {
    %0 = pinch.constant {dst = "a"} 42
    %1 = "pinch.borrow_mut"(%0)
        {dst = "b", src = "a"}
        : (ui32) -> memref<1xui32>
    %2 = "pinch.borrow"(%0)
        {dst = "c", src = "a"}
        : (ui32) -> memref<1xui32>
    pinch.return {src = ""}
  }
}
```

The borrow checker can easily track the referencing of an owned value. The "pinch.borrow_mut" instruction observes that owner 'a' was named as the source for this operation, and it marks 'a' as having a mutable reference loaned out. When the following "pinch.borrow" instruction observes the same source owner 'a', it will also check that there are no mutable references in existence. This check will fail, and will generate the following compiler error:

```
error: Cannot borrow a shared reference while a
mutable reference is active
```

Finally, we will examine a more complicated scenario where the lifetime of a variable is not tied to a lexical scope:

```
fn consume_box(a: Box) -> u32 {
   return *a;
}
fn main() {
   let a = box(2);
   let b = consume_box(a);
   let c = *a;
}
```

This program introduces us to the Box type. The box is a heap-allocated integer, and Pinch needs to free the box when its lifetime ends. This example demonstrates how sometimes the lifetime of owned data is not the lexical scope that data was created in. We create a box, and move it into the 'consume_box' function when we pass it by value. We then try to access the box after it was moved, which is not allowed. Let's take a look at the generated MLIR and see where the "pinch.drop" instruction was generated to free this variable:

```
module {
  func @consume_box(%arg0: !pinch.box)
    -> ui32 attributes
    {src = ["a"], sym_visibility = "private"}
  {
    "pinch.drop"(%arg0) : (!pinch.box) -> ()
    %0 = "pinch.deref"(%arg0)
        {dst = "return", src = "a"}
        : (!pinch.box) -> ui32
    pinch.return %0 : ui32 {src = ""}
  }
  func @main() attributes {src = []} {
    %0 = "pinch.box"()
        {dst = "a", value = 2 : ui32}
        : () -> !pinch.box
    %1 = "pinch.move"(%0)
        {dst = "", src = "a"}
        : (!pinch.box) -> !pinch.box
    %2 = pinch.generic_call @consume_box(%1)
        {dst = "b"}
        : (!pinch.box) -> ui32
    %3 = "pinch.deref"(%0)
        {dst = "c", src = "a"}
        : (!pinch.box) -> ui32
    pinch.return {src = ""}
  }
}%
```

Notice that "pinch.drop" was generated in ’consume_box’ instead of in the main function. Pinch detected that the box was moved out of main, and that the box was not moved out of ’consume_box’. Since the box was moved, it is no longer available in this scope, and our attempts to dereference it will be met with the following error generated by the borrow checker:

```
error: Trying to dereference already moved variable
```

The concept of an ownership graph makes this example possible. Because we are keeping track of what data belongs to what owner at operation-level granularity we can perform effective static analysis on the program. As long as the programmer obeys a few simple rules mandated by our borrow checker we can generate safe code and automatically generate drop operations to free data.

7 MLIR DEVELOPMENT

In hindsight, MLIR allowed for an incredible turnaround time on this project. Reusing existing dialects and the LLVM infrastructure allowed a single developer to completely implement a borrow checked language in 7 weeks. Although the core contribution of this project is the borrow checker, significant work was required to build the frontend and the lowering passes. The lowering passes in particular required extra effort to manage types and memory allocations without hurting performance. As a result, we highly recommend MLIR be used for any future projects such as borrow checking which involve high level static analysis at compile time.

8 RELATED WORK

Rust [1] is by far the most popular language with built in borrow checking. It is widely used in industry and has seen constant development for many years. It is built on LLVM, and has two levels of higher IR that are Rust-specific. Rust also supports complicated lifetime tracking that is not related to the lexical scope of a variable. Rust was one of the motivating examples for the creation of MLIR.

Other borrow checked research languages exist, although they see significantly less use than Rust. Carp [4] is a research language similar to Lisp. Carp introduces many Rust-like features such as static type inference and ownership tracking. Dyon [3] is another small language which was designed to be an interpreted version of Rust. It interfaces with Rust code and follows similar syntax and borrow checking rules.

9 LESSONS LEARNED

This project gave me significantly more experience working with LLVM, and I can’t say I enjoyed it. This is mostly due to the fact that I am not well-versed in complex C++ features, and I had to learn as I went. The majority of my time was not spent working on designing my own MLIR, but reading LLVM code trying to learn what format it expected data in. I walked away with a massive appreciation for the infrastructure that LLVM has built. It may not be an easy code base to learn, but it is incredibly impressive.

10 CONCLUSION

In this project we demonstrate a reference implementation of borrow checking in MLIR using analysis of an ownership graph. We document the steps taken, from AST creation to code generation, and show how MLIR can promote code reuse. Based on our findings, we argue that any new language interested in implementing a borrow checker should do so utilizing MLIR.

REFERENCES

[1] Cosmin Cartas. 2019. Rust – The Programming Language for Every Industry. *ECONOMY INFORMATICS JOURNAL* 19 (09 2019), 45–51. https://doi.org/10.12948/ei2019.01.05
[2] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore’s Law.
[3] Sven Nilsen. 2020. Dyon. https://github.com/pistondevelopers/dyon
[4] Erik Svedäng. 2020. Carp. https://github.com/carp-lang/Carp