CSC 766 DSL
Austin Shafer

Pinch: A simple borrow-checked language

One of the coolest languages emerging in recent years is Rust, which statically enforces memory safety and concurrency and is becoming a very popular systems language. Rust uses custom high and medium level IR on top of LLVM IR, which is exactly the kind of logic MLIR was designed to consolidate. My domain specific language, named Pinch, will be a borrow-checked language similar to Rust which will be built using MLIR.

The language will be very light on features, it will only have basic functions and one integer data type. “Pointers” will be implemented in a similar manner to Rust’s references: multiple shared references may exist but only one mutable reference may. The real contribution of the project is implementing lifetime tracking in MLIR. The compiler needs to track the scope that data is available for, and which named variable owns access to that data. This is an advertised ability of MLIR, but I can’t find any references to anyone actually doing it.

(Some minor details of this language may change during development, but the major concepts will remain the same)

A small sample program will look something like this:

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

fn main() {
        let localvar = named_function(2);
        print(localvar);
}
```

The above program simply multiplies the value passed to named_function by 2 and prints it out.

Function signatures all follow this form, followed by a C-style bracketed block of code.

```
fn function_name(arg_name: type, …) -> return_type
```

The return tag (-> return_type) is simply omitted for a void function. To narrow the scope of the project, the only data type supported is a 32-bit unsigned int, designated by the name u32.

There is only one builtin function, print. Print is defined as such:

```
fn print(arg: u32)
```

Print is included for simple program output. The basic arithmetic operands are supported (+, -, *, /) for manipulating u32. Functions are called in the conventional manner.

A lexical “scope” is designated by a code block bound in brackets (like most “C-style” languages). A variable declared in a scope can only be accessed by that scope. Its value is only available for that scope (along with any subscopes that reference it).

A lifetime refers to the scope that a variable is active for:

```
{
        let v1 = 2;          ← variable v1 is created, and its lifetime begins
        let v2 = v1 * 2;     ← variable v2 uses v1 during its valid lifetime
}                            ← The scope ends, and v1 and v2's lifetimes are concluded
```

In Rust when a lifetime ends the value is dropped. This frees any resources and cleans up the value without using any garbage collection. Because the only data type in this language is u32, which does not own any memory, no extra deconstructors need to be called.

The last major feature is ownership. All data is owned by exactly one variable. Data is also referred to as the value of the variable. A variable's value can be accessed through a pointer by “taking a reference” of the variable. This is called “borrowing” or “referencing” the variable. It has a simple set of rules:

- One variable owns a value in its lifetime
- References must only exist for a subset of the lifetime of the owner they reference.
- Multiple shared references can be made from the owning variable. (These are read-only)
- A mutable reference can be made from the owning variable if and only if there are no other references (shared or mutable).
- While references are handed out, the owning variable cannot modify the value
    - This would allow for pointer aliasing like in C. This is not safe, and is what we are trying to prevent.
- References are valid for the scope they are created in, after which the reference ceases to exist. (once again allowing the owner to access the value)

References are taken with the & operator, and are dereferenced with the * operator. References can be taken in the two ways (shared and mutable) mentioned above:

- Shared reference: &target_var
- Mutable reference: &mut target_var

In both cases the & shows that we are taking a reference to target_var, but we add the mut keyword if we want a mutable reference.

Let’s look at an example:

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

fn main() {
	let v1 = 2;
	let v2 = named_function(&v1);
	print(v2);
}
```

This example is changed to have named_function accept a reference instead of a value. A local variable v1 owns the value, which is “loaned out” in the form of a shared reference to named_function. This reference’s lifetime is the scope of named_function. This is a valid reference because (1) v1 does not modify its value while the reference is alive and (2) the lifetime of the reference does not outlive v1’s lifetime. If you are familiar at all with Rust this will seem very commonplace.

If there are any violations of these rules of ownership, then the compiler will emit error messages and refuse to compile. At the moment it is unclear how effective MLIR is at generating error messages for a specific line of code, so I am not sure how verbose I will be able to make these messages. Either way, the tests for this language will contain both valid code examples and invalid examples that will not compile.

I believe this will be a good project for this class as it exposes me to the complicated intricacies of modern language implementation. Depending on if any others have used MLIR for borrow checking, it may actually be a novel contribution to the field.