Pitfalls in Parallel Programming Language Design

AREN MERZOIAN, University of Twente, The Netherlands

Multithreading is a very important concept for optimizing certain algorithms, the ability to split up certain workloads into isolated chunks and run an algorithm on these chunks across multiple threads can affect the performance of these algorithms by several factors.[13]

In this paper, a programming language called Twice is introduced. Twice is a minimal compiled language that implements multithreading through future[2] (or promise) values. Twice is built on top of the LLVM project[9] to support compilation to many targets and was created to be a learning tool for future language designers. Twice also implements other non-trivial features like top-level statements, control flow through if statements and while loops, a simple type system and access to foreign functions.

This paper will explore the overall development process of the language compiler, discuss issues that came up during development and provide general pointers on what (not) to do when designing a programming language.

Additional Key Words and Phrases: programming language, compilers, LLVM, multithreading

1 INTRODUCTION

Concurrency and parallelism are two interconnected terms[8]. Parallelism describes the ability of a program to perform independent operations at the same time, whereas concurrency is a more general term for the ability to manage blocks of code that may be executed independently. Parallelism implies concurrency, yet the reverse is not necessarily true.

Including concurrency features in a programming language is not a trivial task[1]. This paper explores the creation process of a programming language with parallelism, called Twice. Twice is a minimal strongly-typed compiled language with a simple concurrency model.

This paper aims to answer the following research question through the creation of Twice:

RQ: What are common pitfalls in the implementation of a programming language supporting parallelism?

To aid in answering this research question, it can be split up into three sub-questions.

RQ1: What language features should be implemented for Twice's concurrency model?

RQ2: What problems can occur at the compiler level for a language with this concurrency model?

RQ3: How does the compilation target affect the availability of this concurrency model?

First, the concurrency models of various other programming languages will be discussed in order to define a fitting concurrency model for Twice, and thus, answer **RQ1**. Then, the language features available in Twice will be discussed with short examples for each of these features.

Following this, implementation details of some non-trivial language features are covered to answer **RQ2**. Finally, there will be a short section about platform support to answer **RQ3**. In the appendices, larger programs are shown in Twice.

2 CONCURRENCY IN PROGRAMMING LANGUAGES

There is no single correct way to implement concurrency or parallelism, programming languages vary in their supported concurrency features and the implementation details[4]. To decide how concurrency will be implemented in Twice, it is important to consider the concurrency features available in other programming languages.

2.1 JavaScript and TypeScript

JavaScript and TypeScript allow concurrency through futures[2] (called Promises in the language). Futures can be understood as an implied contract when calling a function. The function will, upon finishing execution, return a value of a certain type. The caller can wait until the function has finished executing whenever this value is needed but can continue executing its own code in the meantime. The JavaScript/TypeScript runtime keeps track of a queue of tasks, and whenever a task finishes execution or needs to wait for another task to finish first, the runtime will start execution of another task instead[3]. JavaScript and TypeScript clearly support concurrent programming, however, they do not allow for parallelism (as with most other interpreted languages). There is always only one task executing, and that task runs for as long as possible.

2.2 Python

An interpreted language that seems, at first glance, to have implemented parallelism is Python (only the standard CPython implementation will be discussed). Python has the multiprocessing package which allows for process-based parallelism[6]. A new Python interpreter process is launched, running the specified function in parallel. Function arguments are serialized to a stream of bytes using the pickle package[7] and passed to the new process to be deserialized back into Python objects and used. Return values work similarly, the new process serializes the return value and passes it back to the main process which can deserialize it. It is also possible to have shared memory between these processes through pickle serialization. This achieves true parallelism, however, it also has a very large cost: the startup of an entirely new operating system process, along with the serialization of parameters, return values, and shared memory. This means that using multiprocessing for trivial tasks can easily end up being slower than doing the task normally.

2.3 Java and C#

Instead of using multiple processes, parallelism can also be achieved through *multithreading*. Languages like Java and C# compile to a platform-independent format that can be run by their runtimes (the JVM and .NET runtime respectively). Parallelism in these languages can be achieved by spawning new threads. Threads are more closely

TScIT 41, July 5, 2024, Enschede, The Netherlands

coupled to each other and have a smaller (but still significant) startup cost when compared to processes[12]. This also means passing data across threads is much simpler compared to processes.

2.4 Go

In situations where the programmer is unsure whether the task that can be separated warrants creating a new thread, a feature from the Go programming language is very useful[10]. Go's concurrency model revolves around the concept of a "goroutine". The programmer can separate out their independent tasks into goroutines, and the Go scheduler decides whether to spawn new threads for these tasks, execute them consecutively, or a mix of both. Furthermore, Go offers a language feature called "channels" which can be used to transfer data back and forth between these goroutines.

2.5 C

In C, the programmer has a lot of explicit control over how concurrency can be managed. Programs targeting UNIX-like platforms can make use of the pthread library[11]. This library makes it possible to execute a function in a new thread executing in parallel and wait for the thread to finish when the result is needed.

OpenMP is another popular option for parallelizing on UNIX-like platforms[5]. OpenMP specifies various compiler directives that allow the programmer to easily parallelize certain code structures. For loops, for example, could be parallelized by simply placing an OpenMP pragma (like #pragma omp parallel for) before the loop in certain cases.

Programmers targeting Windows platforms can make use of functions like CreateThread from the standard library to achieve functionality similar to pthread. OpenMP support on Windows is also available, however, more limited.

3 THE TWICE LANGUAGE

For Twice, the decision was made to implement a concurrency model that resembles Promises in JavaScript because of the ease of use of this feature. In Twice, a function can be marked as "threaded", which implies the function will run in the background in a new thread upon being called. The function returns a Promise type which contains a subtype (or void). When execution finishes, the function should return a value of that subtype.

Unlike JavaScript, these are real threads and offer parallelism, however, they also come with the startup cost of a thread which is more significant than the event loop mechanism in JavaScript. The programmer needs to be aware of this and avoid using threaded functions for small tasks, as the cost of starting a thread may outweigh the benefit of parallelism.

Twice is designed to have minimal syntax while still supporting features that allow the implementation of a variety of programs. Features of Twice include, but are not limited to: functions (and recursion), while loops, if statements, variables (and shadowing), and mathematical operations. It does not include features like arrays, iterators, and classes/objects.

This means that Twice should not be used as a programming language for production applications. It is a learning tool made for language designers who want to target LLVM for their language and want to include concurrency in their language. It can be seen as a starting point or a reference for how language features could be implemented.

3.1 Types

Twice is a strongly typed language and supports a few basic types, promises that may contain another type, and function types that describe the types of all function parameters and the type of the function return value.

The following is a list of all supported types in Twice, the name of the type in the language is italicized.

- Twice supports 32-bit signed integers allowing users to specify integers between -2147483648 and 2147483647 and use them for calculations. The type is called *int*.
- Twice supports 64-bit floating point numbers (called a *double*). Mixing doubles and integers in calculations is not allowed.
- Twice supports strings, a *string* is represented internally as character arrays.
- Twice supports booleans, along with boolean operations like AND (&&) or OR (||). The type is called *bool*.
- Twice supports *promise* values, they have a dual purpose as they represent a handle to another thread, and contain information about what type the function running in that thread will return. It is possible to wait for the thread to finish and obtain the return value using the await statement.
- The type of a *function* can also be specified. This has the use case of specifying parameters and return types when importing an external function through LLVM (like printf). It's possible to import any function that can be described entirely using types supported by Twice. Functions that do not return a value can be described using void as the return type.

The decision to use 32 bits for integers and 64 bits for doubles came from a desire to keep the initial prototype of the language simple. Initially, Twice used 32-bit integers and floats, however, to support calling variadic functions doubles were chosen instead. This is because of C promoting floats to doubles in these cases, using doubles ensures types line up more often.

3.2 Statements

Statements are the core of a Twice program, they represent an action that affects the state of the program, like storing a value in a variable or taking action based on the current value in a variable. There are many types of statements.

3.2.1 Variable declarations and assignments. Twice allows users to define *variables*, store values in them, and reassign them later.

```
let x = -5; -- The value of x is -5
```

Listing 1. It is possible to declare a variable and store a value inside of this variable using the let keyword

In the above example, a new variable x is created which contains the value -5. The value of x can be changed through reassignment.

```
let x = -5; -- The value of x is -5
x = 3; -- The value of x is now 3
```

Listing 2. It is possible to reassign the value of x

```
let x = -3;
let x = -5; -- Not allowed, x was already defined in this
    scope.
```

Listing 3. It is not possible to redeclare a value in the same scope

```
let x = 3;
-- The value of x is 3 here
{
    let x = "Hi!";
    -- The value of x is "Hi!" here
}
-- The value of x is 3 here again
```

Listing 4. It is possible to redeclare x in a new scope

```
let x = 5; -- The type of x is int
x = "Hi!"; -- Not allowed, assignment of a string to an
    int variable.
```

Listing 5. Assigning a value with an invalid type

3.2.2 Blocks. Blocks are groups of statements that will execute consecutively. Blocks create a new scope for variable information, and upon leaving the scope these variables are not accessible anymore. Variables from higher scopes are accessible, but may also be shadowed with new declarations

```
let x = 3;
-- The value of x is 3 here
{
    x = 5;
    -- The value of x is 5 here
}
-- The value of x is still 5 here
```

Listing 6. Modifying a variable in an outer scope

```
let x = 3;
-- The value of x is 3 here
{
    let x = 5;
    -- The value of x is 5 here
}
-- The value of x is back to 3 here
```

Listing 7. Shadowing

3.2.3 Functions. Functions are reusable pieces of code that can take input parameters and may produce a return value based on these parameters. A function always has a *signature*, specifying the name of the function, the type of all parameters, and the type of the return value.

Twice allows the programmer to include external functions from a different LLVM module (like the C standard library), this can be done using the *extern* statement and requires knowing the function's signature. The signature for printf specifies *vararg*, which means that it can take additional optional arguments. Furthermore, printf specifies the return value *void*, which means it does not return any value. It has one required parameter, which must be a *string*. The following is an example of an extern statement for printf.

```
extern vararg fn printf: void(string);
```

Listing 8. Including printf from the C standard library

Functions can be called, which executes the code in the function with the given parameters.

```
extern vararg fn printf: void(string);
printf("Hello, World!");
```

Listing 9. Hello, World! in Twice

Functions can also be defined by the programmer, through a function definition.

```
extern vararg fn printf: void(string);
fn addOneAndPrint(x: int): void {
    printf("your number: %d", x + 1);
}

addOneAndPrint(4); -- your number: 5
```

Listing 10. Function definition example

Functions may return values using the *return* statement, the caller can then use this value.

```
extern vararg fn printf: void(string);

fn justAddOne(x: int): int {
    return x + 1;
}

printf("return value: %d", justAddOne(4));
```

Listing 11. Function return values

Functions can also be marked as threaded, which will make them run in the background in a new thread. Threaded functions must return a value wrapped in a promise, however, this may be void. Threaded functions may not affect variables outside of the function, all input must be passed through parameters.

The following example is an incorrect use of threaded functions, a variable is declared outside the threaded function and the threaded function attempts to read from and modify the variable. Twice does not allow this behavior, as it can often lead to unpredictable programs. In this case, it is impossible to predict whether the value of x will be 4 or 5.

```
extern vararg fn printf: void(string);
let x = 4;
threaded fn waitAndAdd(): promise<void> {
    x = x + 1; -- Not allowed, x is declared outside
}

waitAndAdd();
printf("x: %d", x);
```

Listing 12. Incorrect example, bypassing promise system

The programmer needs to choose whether they want to wait for the thread to finish and get a return value, or use the value before it was passed to the function.

```
extern vararg fn printf: void(string);
let x = 4;

threaded fn waitAndAdd(x: int): promise<void> {
    x = x + 1; -- Overwrites a copy of x
}

await waitAndAdd(x);
printf("x: %d", x); -- This will always output x: 4
```

Listing 13. Correct example, x is 4

```
extern vararg fn printf: void(string);
let x = 4;

threaded fn waitAndAdd(x: int): promise<int> {
    return x + 1;
}

printf("x: %d", await waitAndAdd(x)); -- This will always
      output x: 5
```

Listing 14. Correct example, x is 5

The above example uses the *await* statement to wait until the thread finishes execution and obtain the return value from the function. This guarantees that the function will always execute entirely, and is the only way to obtain the return value from a threaded function.

3.2.4 If statements. Twice supports *if-then-else statements*, this allows the programmer to only execute a certain block of code in some cases. Optionally, an *else* block can be included. If the condition holds, the *then* block gets executed, otherwise, the *else* block gets executed if it is specified. These statements can also be chained together.

```
extern vararg fn printf: void(string);
let myNumber = 15;

if (myNumber % 3 == 0 && myNumber % 5 == 0) {
    printf("FizzBuzz");
} else if (myNumber % 3 == 0) {
    printf("Fizz");
} else if (myNumber % 5 == 0) {
    printf("Buzz");
} else {
    printf("%d", myNumber);
}
```

Listing 15. A simple one number FizzBuzz example

When returning a value with an if statement, the else path is required and must return a value of the same type. This limitation is further explored in section 6.1.

```
fn test(): int {
    if (false) {
        return 4;
    } else {
        -- Not allowed, this path must return an integer
    }
}
```

Listing 16. Non-matching return types in if statement

3.2.5 While statements. The final statement in Twice is the *while* statement. This allows the programmer to repeatedly execute a block of code while a certain condition holds. A typical use case of a while loop is to implement a counter variable.

```
extern vararg fn printf: void(string);

let x = 1;
while (x <= 100) {
    printf("%d\n", x);
    x = x + 1;
}
```

Listing 17. While loop to implement a counter

3.3 Expressions

Expressions represent a value in Twice: whether that be a literal, the result of a calculation, the return value of a (threaded) function, or the value stored inside of a variable.

3.3.1 Literals. Most types allow you to create a value through a literal. Literals are simple expressions that represent a constant value.

```
let intLiteral = 3;
let otherInt = -213;
let floaty = 1.2;
let otherFloaty = 15.0;
let boolValue = true; -- or false
let myName = "Aren";
```

Listing 18. Various examples of literals

3.3.2 Mathmetical operations. Twice supports several mathematical operations for integers and doubles. Integers and doubles may not be mixed in mathematical operations.

```
let add = 1 + 2; -- Addition, 3
let addWrong = 1 + 2.0; -- Not allowed, int and double
let sub = 1 - 2; -- Subtraction, -1
let mul = 1 * 2; -- Multiplication, 2
let div = 4 / 2; -- Integer division, 2
let divFloat = 1.0 / 2.0; -- Float division, 0.5
let mod = 15 % 4; -- Modulo/remainder, only for ints, 3
```

Listing 19. Supported mathematical operations in Twice

Twice also allows for the comparison of two numeric values. A comparison always results in a boolean value. Once again, these numbers must be of the same type.

```
let lt = 2 < 1; -- Less than, 0 (false)
let lte = 2 <= 2; -- Less than or equal, 1 (true)
let gt = 2 > 1; -- Greater than, 1 (true)
let gte = 2 >= 3; -- Greater than or equal, 0 (false)
let eq = 3 == 3; -- Equality, 1 (true)
let neq = 3 != 3; -- Inequality, 0 (false)
```

Listing 20. Comparisons in Twice

Equality and inequality can also be used on booleans, as well as several boolean operators.

```
let or = false || false; -- Boolean OR, false.
let or2 = true || false; -- true
let or3 = true || true; -- true
let and = false && false; -- Boolean AND, false.
let and2 = true && false; -- false
let and3 = true && true; -- true
let neg = !true; -- false
let neg2 = !false; -- true
```

Listing 21. Boolean operators

3.3.3 Variables. A variable may be referenced as an expression, the value represents the content of the variable

```
extern vararg fn printf: void(string);
let x = "Hello!";
printf(x); -- Hello!
```

3.3.4 Function calls. Function calls can also be expressions when the function has a return value. Threaded functions always return a promise, and *await* may be used to wait for the function to complete and obtain the return value (where relevant).

```
extern vararg fn printf: void(string);
fn myFunc(y: int): int
    y + 1

let x = myFunc(4);
printf("x: %d", x); -- x: 5

threaded fn myTFunc(y: int): promise<int>
    y + 1

printf("threaded: %d", await myTFunc(5));
```

While this initial version of Twice definitely has its limitations, this set of features still shows a powerful language. In section 6, features that did not make it into the initial version of Twice are discussed. These features could be implemented in the future.

4 IMPLEMENTATION DETAILS

During the creation of Twice some language features proved to be difficult to implement, this paper covers these difficulties and the solution that was found for each of these features.

4.1 Type checking

The Twice compilation process consists of several steps:

(1) Tokenizing the input and parsing tokens into a parse tree
(2) Transforming the parse tree into an Abstract Syntax Tree (AST)
(3) Performing type analysis on the AST
(4) Generating LLVM bitcode based on the AST and type information
(5) Compiling LLVM bitcode into native binary.

For step 3 specifically, a type checker has to be implemented. The type checker walks down the AST and defines the type of a node based on its children.

The type of an expression is simply the type of the value it represents. Adding two integers together should result in an integer, comparing whether an integer is larger than another should result in a boolean value.

Additionally, expressions make sure the types of their child elements make sense. It's not possible to use boolean operators like && on strings, for example. These invalid expressions result in type errors and stop the program from compiling.

The type of a statement in Twice represents the ability of this statement to return a value. This is useful for determining whether the return value of a function matches the function's signature, for example.

4.1.1 LLVM types. The type checker must store the type of an expression or statement using some form of data structure. Twice uses a dictionary from AST nodes to internal LLVM types.

The benefit to this is that every AST node has a corresponding LLVM type and this reduces the complexity of the IR generation step later.

The downside to this is that LLVM types can be quite limited. In newer LLVM versions, pointers do not store information about the type of value they point to. This meant that certain types, like arrays, would prove to be very difficult to implement without major changes to the type checker.

Because of the restriction to LLVM types, the promise type in Twice was implemented in a unique way, it is stored as an LLVM *struct* containing a single value (a handle to the thread). As structs are otherwise not used in Twice as expressions, this does not conflict with any other Twice features. This struct is only visible to the type checker, and will not appear in generated code. This is a workaround, and ideally, the type checker would be able to work with promise types. A reimplementation of the type checker that would allow this to work more nicely is explored in section 6.1

4.1.2 Lexical scoping. Variables in Twice are scoped and can be shadowed. This means that they must be stored in a data structure that can contain two values with the same name in some cases, but not in other cases.

Twice uses a data structure internally called a Variable Stack for this, it is defined with two fields, a dictionary mapping from strings to LLVM types, and, optionally, a reference to a parent Variable Stack. When a new scope is entered, a new Variable Stack is created with the current Variable Stack set as its parent. When a scope is left, the parent Variable Stack becomes the current Variable Stack.

When Twice tries to look up information about a variable, it checks the current Variable Stack to see if the variable can be found. If not, it recursively checks the parents to find the first instance of the variable.

When a variable is declared, it must not already exist in the current Variable Stack, as this would imply two declarations in the same scope. It may, however, appear in one of the parent Variable Stacks.

During IR generation, a data structure similar to Variable Stacks is used. However, instead of containing LLVM type information, they contain a reference to the location of the variable.

4.2 Functions and top-level statements

Functions in LLVM consist of interconnected basic blocks that contain instructions, these basic blocks must end with special *terminator* instructions that indicate the end of the block. A block must have exactly one terminator instruction, and may not have regular instructions after the terminator instruction. During code generation, the Twice compiler keeps track of the current basic block being worked on. Instructions can not appear outside of a basic block (and thus, a function).

According to this model, statements in Twice would always belong to a function. However, it is clearly possible to have statements in Twice that are not part of a function (called *top-level statements*) as shown in prior listings. Twice works around this problem by creating a *main* function that contains the top-level statements during code generation.

When a new function is defined, a new basic block is created for this function too. During code generation for this function, this new basic block is used as the current basic block. Once the function definition ends, the previous basic block is restored for further code

generation. This means Twice can interrupt code generation for the main function to start generating code for a different function, and go back to the main function afterwards.

4.3 Control flow

Twice allows the programmer to modify the flow of the program through if statements and while loops. These features are implemented by connecting basic blocks together in LLVM. It is important to keep in mind all basic blocks must have *exactly one* terminator, control flow has to be implemented with this restriction.

4.3.1 If statements. In an if statement, at least 2 basic blocks are used. When the if statement also includes an else portion, an additional third basic block is required. The basic blocks have the following tasks:

- Basic block 1 (*then* block) contains code that executes when the condition is true. At the end of the code, a branch terminator is placed that will always continue execution at basic block 2. If the code inside of basic block 1 already generates a terminator (because of an early return, for example), this branch terminator is not included.
- Basic block 2 (*after* block) contains code that is executed after the if statement. The if statement does not generate any code inside of this block, it is simply marked as the next primary block for code generation after the if statement.
- Basic block 3 (*else* block) is only used when the if statement includes an else portion. It is identical to basic block 1 and behaves similarly, but runs when the condition is false, not true.

In the original basic block, code is inserted to jump to basic block 1 when the condition holds. When the condition does not hold, code is inserted to jump to basic block 3 if an else portion is present, or basic block 2 otherwise.

The following is an example of an empty if statement that always jumps to basic block 1 in LLVM IR.

```
define void @ifTest() {
entry:
  br i1 true, label %thenBlock, label %elseBlock

thenBlock:                ; preds = %entry
  br label %afterBlock

elseBlock:                ; preds = %entry
  br label %afterBlock

afterBlock:               ; preds = %elseBlock, %thenBlock
  ret void
}
```

Listing 22. Code generation for empty if statement in IR

4.3.2 While loops. While loops always use three basic blocks in Twice.

- Basic block 1 (*check* block) is used for verifying whether the statement holds or not. It generates code for an expression that results in a boolean value and jumps to basic block 2 if this boolean is true or basic block 3 if this boolean is false.
- Basic block 2 (*loop* block) is used to contain the statements that are inside of the while loop. A branch terminator is placed at the end of these statements to jump back to basic block 1 unless a terminator is already present.
- Basic block 3 (*after* block) behaves similarly to basic block 2 in an if statement. The while statement does not generate any code inside of basic block 3, rather, it is used as the next primary block for code generation.

The following is an example of a while loop with a variable counting from 0 to 99. IR used for generating the variable is not included in this example.

```
define void @whileTest() {
entry:
  ...           ; %x is allocated and initialized with 0
  br label %check

check:                            ; preds = %entry, %loop
  %0 = load i32, ptr %x, align 4
  %1 = icmp slt i32 %0, 100
  br i1 %1, label %loop, label %after

loop:                             ; preds = %check
  %2 = load i32, ptr %x, align 4
  %3 = add i32 %2, 1
  store i32 %3, ptr %x, align 4
  br label %check

after:                            ; preds = %check
  ret void
}
```

Listing 23. Code generation for simple counter in IR

4.4 Threaded functions

To support threaded functions, Twice makes use of POSIX threads[11]. Twice programs automatically include both the *pthread_create* and *pthread_join* functions. These library functions are used internally by the Twice compiler to generate code for threaded functions.

Functions that can be used with pthread need to have a specific signature, they may only accept a single pointer as a parameter and can return a pointer. Twice allows threaded functions to accept multiple arguments, and thus, needs to work around this issue.

When creating a threaded function, instead of passing the parameters to the function through normal function parameters, the Twice compiler passes function arguments through a single struct. The types contained in the struct depend on the function parameters. A pointer to this struct can then be used as an alternative to the regular function parameters.

Instead of directly returning a value, the start of a threaded function allocates memory for the return value. When a return statement is encountered, instead of generating a *ret* terminator and ending the function, it first writes the return value to this memory location. It then always uses *ret* to return a pointer to this memory location. This works around the issue of always having to return a pointer. When *await* is used, the value this pointer points to is read to retrieve the actual return value.

The call to *pthread_create* returns a handle to the thread. Twice represents this handle internally using the promise type. Await statements or expressions can then pass this handle to *pthread_join*,

this will block the current thread until the execution of the other thread is finished. Following this, *pthread_join* returns the pointer to the return value. Twice can then load the value stored in the pointer and use it.

5 SUPPORT FOR MULTIPLE PLATFORMS

The Twice compiler is written in C# using .NET 8.0, with the only dependencies being ANTLR for parser generation and LLVMSharp for the LLVM bindings. This means that the Twice compiler itself can run on many platforms without issue.

Twice programs using threaded functions additionally depend on libpthread (or an equivalent implementation) during runtime. macOS and Linux platforms widely support pthread, on Windows libwinpthread is available through Cygwin and must be placed on the PATH.

Additionally, depending on what external functions are included in the Twice program, more modules might have to be linked in during compilation. Depending on the functions, this may break support for certain platforms. The programmer additionally has to make sure the module is linked in during the final step of compilation with clang.

6 FUTURE WORK

For the initial version of Twice, there were certain planned changes and missing features that would have fit well into the language but could not be implemented for various reasons.

6.1 New type checker

The type checker in Twice can only store limited type information, and, as such, can behave strangely in certain situations where a clear type can not be discerned. An example of this is the if statement, the type checker will allow the following code.

```
let x = 4;
fn test(): int {
    if (x == 3)
        return 1;
}
```

Listing 24. Non-exhaustive return from function

This code is allowed through the type checker, as the type checker does not have a way of specifying a "possible" return and won't fail. Either a statement returns a value, or it doesn't return. In this case, the type checker assumes the if statement will always return an integer. This is not a major problem as code generation will fail because one of the blocks in the function does not terminate, however, ideally, the type checker should be able to find this problem and give an error.

Furthermore, some features could not be implemented like support for arrays and iterating them using *for* loops. Code generation for arrays could not be implemented, as there is not enough type information to access an element from an array. Arrays are stored as pointers to the first value and contain information about the length. However, LLVM does not store the type a pointer resolves to, and as such, during code generation the type of a value inside an array is unknown. Implementation of a new type checker would allow arrays (and other data structures) to be added to Twice.

6.2 Additional concurrency features

Due to setbacks during development, implementation for the concurrency features in Twice started late in the development cycle. There were some additional concurrency features planned that did not make it in the initial version of Twice, like channels.

Channels would have allowed the programmer to pass values of a specific type between threads. A channel of a specific type could be created before calling a threaded function and then passed to that function as a parameter. Channels are intended for communicating with threads that should run for a long time, where parameters and return values do not fit the requirements.

```
threaded fn add3(channel: channel<int>): promise<int> {
    let c = 0;
    let sum = 0;
    while (c < 3) {
        sum = sum + (<- channel);
        c = c + 1;
    }
    return sum;
}
```

Listing 25. An example of how channels would be used

6.3 Globals

Twice does not have the concept of a global variable, all variables are only accessible to functions they're defined in. Due to the way top-level statements are implemented in Twice, all variables defined in the top-level appear to be global variables, however, they are actually variables local to the *main* function. This means that they are not accessible in other functions.

```
let x = "Hello, World!\n";

fn test(): void {
    printf(x); -- x is not defined
}

printf(x); -- this is allowed
test();
```

Listing 26. Code attempting to use a non-existant global

Ideally, variables declared in top-level statements should be handled differently and represent globals that are accessible across the entire module. Threaded functions should not have access to these globals to prevent many race conditions.

6.4 Testing

Currently, language features in Twice are mostly untested. With the addition of new features to Twice, especially if these features require many changes, it is quite possible that older features stop behaving correctly. To ensure stability, it is important to ensure all language features are tested automatically.

Ideally, a framework would be written to allow full compiler tests. The test specifies Twice source code, its expected output, and any libraries that need to be linked. The framework should then compile the source code, link it appropriately, and then execute the test. Finally, the output is compared against the expected output concluding the test.

Such a framework would allow the addition of new features with a guarantee that the new feature does not break existing tested features.

7 DISCUSSION

Overall, Twice in its current state is a programming language with upsides and downsides. The goal of this research was to explore the difficulties when developing a programming language that includes features related to concurrency and parallelism. Through the implementation of threaded functions (which work using promises), Twice offers a simple concurrency model that lets the user perform tasks (which do not rely on the global state) in the background. All information needed by a task is passed through function arguments, and when the function finishes, it returns its value through a promise.

To clarify on **RQ1**: this is not an all-encompassing concurrency model, it is for example very difficult to implement a thread pool in Twice, as the concurrency model is designed around threads performing a single task where all input is known beforehand. However, the decision was made to go for this simpler model as Twice is designed to be a minimal language, and threaded functions abstract away much of the work for the programmer. Twice initially only supports threaded functions, however, it could be expanded to include other features like channels as mentioned in section 6.2.

Furthermore, regarding **RQ2**, shortcuts and workarounds had to be taken due to time constraints, however, the threaded function system was implemented. During code generation for function calls, it is important the caller knows the type of the return value. It can be looked up through type checker information and can be accessed correctly. For threaded functions, this is more difficult, as threaded functions return a promise. When this promise (internally a pointer) is stored, information about the underlying type is lost due to limitations in the type checker. This means that, later, it is non-trivial to retrieve the underlying type when attempting to *await* a promise. This was worked around by storing a struct instead of a pointer during the type checking phase.

Finally, as for **RQ3**, supporting multiple platforms turned out to be simpler than expected for this concurrency model. Twice's threaded functions represent a very simple use of threads, they do not share global state and only take in function arguments. This means that the implementation of threaded functions could be done using *pthread*, a lightweight C library that has been ported to many platforms. With this being the only required dependency, simple Twice programs are just as portable.

8 CONCLUSION

Designing a programming language, especially when supporting concurrency, is a goal that requires a lot of preparation. Having an understanding of useful data structures to maintain during type checking and code generation, knowledge about the specifics of your target, and an understanding of how language features should be implemented at a low level can reduce much of the load during development. The most significant pitfall encountered during the development of Twice was not necessarily related to the implementation of parallelism, nor was it related to other language features. It was, in fact, a much more general issue. A lack of sufficient preparation and research (especially with regard to type checking) limited the initial prototype. Twice had to be entirely rewritten in the middle of the development cycle due to an oversight with regard to library compatibility. Better preparation could have prevented this and would have led to a prototype with a more complete language.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Peter Lammich, for the time and effort put into supporting the project through his advice and feedback during the meetings.

Additionally, while they were not directly involved with the project, I'd like to thank Victor Tran and John Tur from the bits & Bytes community for their advice and debugging expertise.

REFERENCES

[1] Peter A. Buhr and Glen Ditchfield. 1992. Adding Concurrency to a Programming Language. *University of Waterloo* (1992), 207–224.
[2] A. Chatterjee. 1989. FUTURES: a mechanism for concurrency among objects. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing* (Reno, Nevada, USA) *(Supercomputing '89)*. Association for Computing Machinery, New York, NY, USA, 562–567. https://doi.org/10.1145/76263.76326
[3] Node.js contributors. 2024. The Node.js Event Loop. https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick
[4] Silvia Crafa. 2015. The role of concurrency in an evolutionary view of programming abstractions. *Journal of Logical and Algebraic Methods in Programming* 84, 6 (2015), 732–741. https://doi.org/10.1016/j.jlamp.2015.07.006 "Special Issue on Open Problems in Concurrency Theory".
[5] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313
[6] Python Software Foundation. 2024. multiprocessing - Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html
[7] Python Software Foundation. 2024. pickle - Python object serialization. https://docs.python.org/3/library/pickle.html
[8] Dan Grossman and Ruth E. Anderson. 2012. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 505–510. https://doi.org/10.1145/2157136.2157285
[9] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665
[10] Jeff Meyerson. 2014. The Go Programming Language. *IEEE Software* 31, 5 (2014), 104–104. https://doi.org/10.1109/MS.2014.127
[11] Frank Mueller. 1993. A Library Implementation of POSIX Threads under UNIX. In *USENIX Winter*. https://api.semanticscholar.org/CorpusID:702400
[12] Oracle. 2014. Processes and Threads. https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
[13] M Shanthi and A Anthony Irudhayaraj. 2009. Multithreading-an efficient technique for enhancing application performance. *International Journal of Recent Trends in Engineering* 2, 4 (2009), 165.

A STATEMENT REGARDING USE OF AI

During the preparation of this work the author did NOT use any form of AI. The author takes full responsibility for the content of the work.

B COMPLETE FIZZBUZZ

FizzBuzz is a simple program that counts from 1 to 100, replacing all multiples of 3 by "Fizz", all multiples of 5 by "Buzz" and replacing all multiples of both by "FizzBuzz", on one line.

A simple version for one number was shown in 3.2.4, however, a full version can be seen below.

```
1 extern vararg fn printf: void(string);
```

```
let x = 1;
while (x <= 100) {
    if (x % 3 == 0 && x % 5 == 0) {
        printf("FizzBuzz\n");
    } else if (x % 3 == 0) {
        printf("Fizz\n");
    } else if (x % 5 == 0) {
        printf("Buzz\n");
    } else {
        printf("%d\n", x);
    }
    x = x + 1;
}
```

Listing 27. FizzBuzz

C PARALLELISM SHOWCASE

A simple program that showcases parallelism in Twice can be implemented using a threaded function that counts to 10000 with printf. Running this function twice will, in most cases, show two interleaved counts going up.

```
extern vararg fn printf: void(string);
threaded fn count(): promise<void> {
    let x = 0;
    while (x < 10000) {
        printf("%d\n", x);
        x = x + 1;
    }
}
let x = count(); let y = count();
await x; await y;
```

Listing 28. Two independent counters to 10000