Code generation and intermediate representation in Ballerina Compiler

April 2024

Lecture Outline

- Evolution of Ballerina Compiler
- Current structure of Ballerina Compiler
- Intermediate representations
  - Types of representations
  - Lowering to IR

Evolution : WSO2's history in languages

WSO2 enables thousands of enterprises, including hundreds of the world's largest corporations, top universities, and governments, to drive their digital transformation journeys—executing more than 18 trillion transactions and managing more than 500 million identities annually.

Apache Synapse

XML based DSL for specifying service mediation.
Community project, heavily contributed by WSO2.

Integration Studio

Graphical editor for Apache Synapse

Jaggery Runtime

JS runtime written in Java.

Evolution of Ballerina Swan Lake Compiler

- Started out as Synapse replacement language back in late 2016. Inspired by sequence diagrams and graphical editing.
- Initial implementation as AST interpreted language (2017)
- Internal vm (BVM) with internal ByteCode (late 2017)
- Backend/frontend separation via BIR. JVM bytecode as the backend (late 2018).
- Swan Lake version GA release in , with major improvements and extensive set of standard libraries and connectors (early 2022).
- Continuous updates to Swan Lake version. Currently on update 8.5

Structure of Ballerina Swan Lake Compiler

Parser

Analysis and Desugar

IR Generate

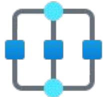
Code Generate

Features of Ballerina

Data oriented

Type-safe, declarative processing of JSON, XML, and tabular data with language-integrated queries.

```
type User record { int id; string name; };
...
User manu = { id: 92874, name: "manuranga" }
```

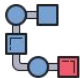


Concurrent

Easy and efficient concurrency with sequence diagrams and language-managed threads without the complexity of asynchronous functions.

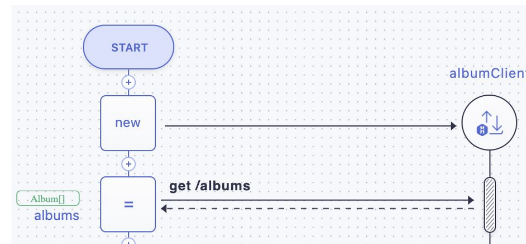
```
http:Client hello = check new ("http://hello.com");
MyGreeting greeting = check hello->get("/world");
```

Also see: start, wait and workers



Graphical

Programs have both a textual syntax and an equivalent graphical form based on sequence diagrams.



Features of Ballerina

Flexibly typed

Uses structural types with support for openness for static typing within a program and for describing service interfaces.

```
type Customer record {|
    int id;
    string name;
    int account;
|};
...
Customer customer = { ... };
User user = customer;
addUser(user);
```

Reliable, maintainable

Explicit error handling, static types, and concurrency safety, combined with a familiar, readable syntax make programs reliable and maintainable.

Cloud native

Network primitives in the language make it simpler to write services and run them in the cloud.

```
> bal build
Compiling source
        example/greeter:0.1.0

Generating executable

Generating artifacts...

        @kubernetes:Service            - complete 1/1
        @kubernetes:Deployment         - complete 1/1
        @kubernetes:HPA                - complete 1/1
        @kubernetes:Docker             - complete 2/2
```

Evolution : Parser

Parser Generator

```
functionSignature :
    LEFT_PARENTHESIS
    formalParameterList?
    RIGHT_PARENTHESIS
    returnParameter?;
```

Easier to get started
Less boilerplate

We used ANTLR until late 2020

https://github.com/ballerina-platform/ballerina-lang/blob/v0.995.9/compiler/ballerina-lang/src/main/resources/grammar/BallerinaLexer.g4

Handwritten Recursive Descent

```
private STNode parseFuncSignature(boolean isParamNameOptional) {
    STNode openParenthesis = parseOpenParenthesis();
    STNode parameters = parseParamList(isParamNameOptional);
    STNode closeParenthesis = parseCloseParenthesis();
    endContext(); // end param-list
    STNode returnTypeDesc = parseFuncReturnTypeDescriptor(isParamNameOptional);
    return STNodeFactory.createFunctionSignatureNode(openParenthesis, parameters,
closeParenthesis, returnTypeDesc);
}
```

Faster for complex grammars
Better at handling edge cases, more flexibility
Better error recovery

It could be much simpler :-
https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing

https://github.com/ballerina-platform/nballerina/blob/main/compiler/modules/front.syntax/parseExpr.bal

Evolution : Execution

AST Interpreter

```
public BValue visit(BinaryExpression binExpr) {
  Expression rExpr = binExpr.getRExpr();
  BValueType rValue = rExpr.execute(this);
  Expression lExpr = binExpr.getLExpr();
  BValueType lValue = lExpr.execute(this);
  return binExpr.getEvalFunc().apply(lValue, rValue);
}
```

Close to source language

ByteCode Interpreter

```
while (ip < code.length) {
 Instruction instruction = code[ip];
 int[] operands = instruction.getOperands();
 int opcode = instruction.getOpcode();
 switch (opcode) {
          case InstructionCodes.ICONST:
          cpIndex = operands[0];
          i = operands[1];
          sf.longRegs[i] = ((IntegerCPEntry)
constPool[cpIndex]).getValue();
          break;
          case InstructionCodes.FCONST:
          cpIndex = operands[0];
          i = operands[1];
          sf.doubleRegs[i] = ((FloatCPEntry)
constPool[cpIndex]).getValue();
          break;
          ...
 }
}
```

Reasonable speed (Order of Python)
Used by Ballerina compiler until 2019

Compiled (to external format)

We are using JVM bytecode in Swan Lake Version

Experimented with LLVM and WebAssemblye

Evolution : Compiler phases

- Less moving parts, simpler

- More people can work code base independently

We started on the lower and keep adding more phases

- Micro pass approach
- Each pass is logical simple

eg: LLVM optimizer.

Parser	We used to have ANTLR, now we have a custom parser
Analysis and Desugar	Create symbols, Type check, Run plugins Lower lambdas and other high level control flow See CompilerPhaseRunner Class of ballerina-lang repo
IR Generate	Create BIR. Conditionals get lowered to GOTOs
Code Generate	Create JVM Bytecode. Create concurrency yield points. Generate classes for values.

Intermediate representation

Q: How are compilers phases are connected? A: IR

Why IR (instead in-memory graph)

- Easy to debug due to serialization
- Can verify

Styles of IR

- Stack based vs Register based
- Flat vs structured
- SSA Register vs Mutable Register

IR Styles : Stack machine vs Register base

```
a = b + 20;
```

Stack machine

```
iload %1
iload 20
iadd
```

- May produce smaller IR.
- Used by JVM and WebAssembly

We didn't pick this option due to the added complexity in generation.

Register base (mutable)

```
iload 20 %2
iadd %1 %2 %0
```

- More closer to source language.

We use this format in Ballerina (BVM, jBallerina IR)

Register base (SSA)

```
%a = add %b 20
```

- Better for analysis and optimization.
- Need phi nodes

We didn't pick because not a good input format for JVM (or for LLVM surprisingly, due to debug info)

IR Styles : Flat vs Structured

Flat IR

```
loopHead:
      local.get $i
      i32.const 10
      i32.gt_s loopEnd
      goto loopHead
loopEnd:
```

- Most popular format
- Can result in non-reducible loops

Structured IR

```
(loop $my_loop
      local.get $i
      i32.const 10
      i32.lt_s
      br_if $my_loop)
```

- Used by WebAssembly
- No GOTOs

Ballerina IR Example

```
public function add(int a, int b) returns int {
    return a + b;
}
```

```
public add function(int, int) -> int {
        %0(RETURN) int;
        %1(ARG) int;
        %2(ARG) int;

        bb0 {
                %0 = %1 + %2;
                GOTO bb1;
        }
        bb1 {
                return;
        }

}
```

- %0 is the return value
- Variables are mutable in general
- Basic blocks are identified. We use this to create safe points for yield.
    - We use 'Duff's device' like approach to compile Ballerina to support user space threading
- IR is typed

Ballerina IR Generation

- Simple recursive depth first walk over the graph
- Users visitor pattern
- Pros : simple, local decision making
- Cons: repeated code
  - Fix: post-code gen cleanup.

https://github.com/ballerina-platform/ballerina-lang/blob/master/compiler/ballerina-lang/src/main/java/org/wso2/ballerinalang/compiler/bir/BIRGen.java

JVM bytecode Generation

- Iterate BIR and generate JVM code using ASM library

- Ballerina supports user space scheduling. We generate a switch that can jump to any BasicBlock from the top to the function. This helps us resume a function (AKA: Duff's device).

- To support Ballerina's structural typing we lower property access to Map.get() but actual implementation of the "Map" can be a specifically generated class

https://github.com/ballerina-platform/ballerina-lang/blob/master/compiler/ballerina-lang/src/main/java/org/wso2/ballerinalang/compiler/bir/codegen/methodgen/MethodGen.java

Take home task

https://github.com/manuranga/ir-gen-uom

Small task to familiarize yourself with ir generation and Ballerina language.
Send us over discord https://discord.com/invite/ballerinalang for a small gift.