LLVM Sauce

A Source to LLVM compiler

Jiachen, Xiaodong

Last revised April 8, 2021

Introduction

Objectives

- To build a Source 1 compliant compiler

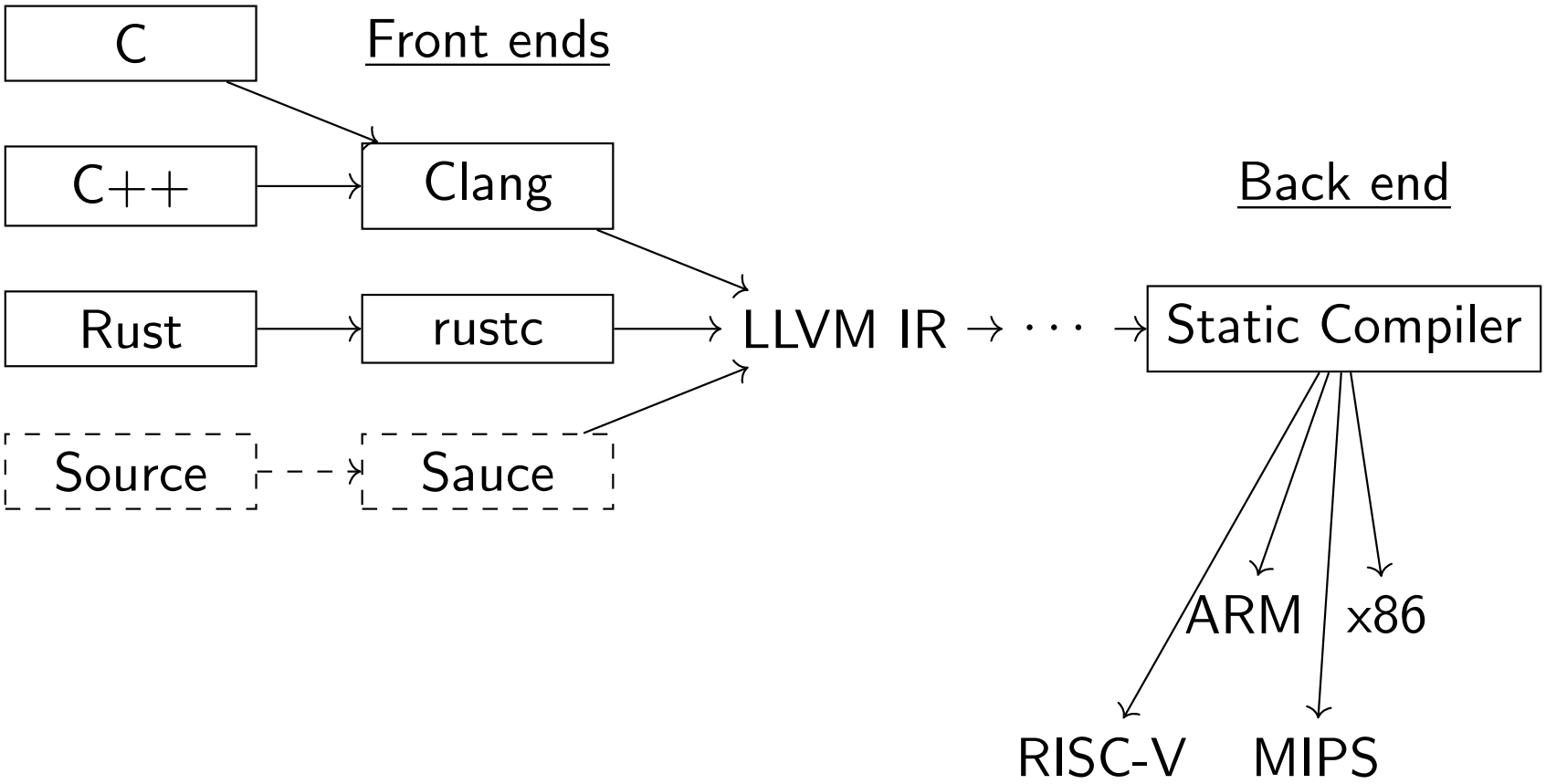
Introduction

Presentation Outline

- Brief intro to LLVM
- Source A — expression language
- Source B — runtime type checking
- Source B— control flow
- Source C— functions
- Source D— tail calls
- Source E— load/store optimisations

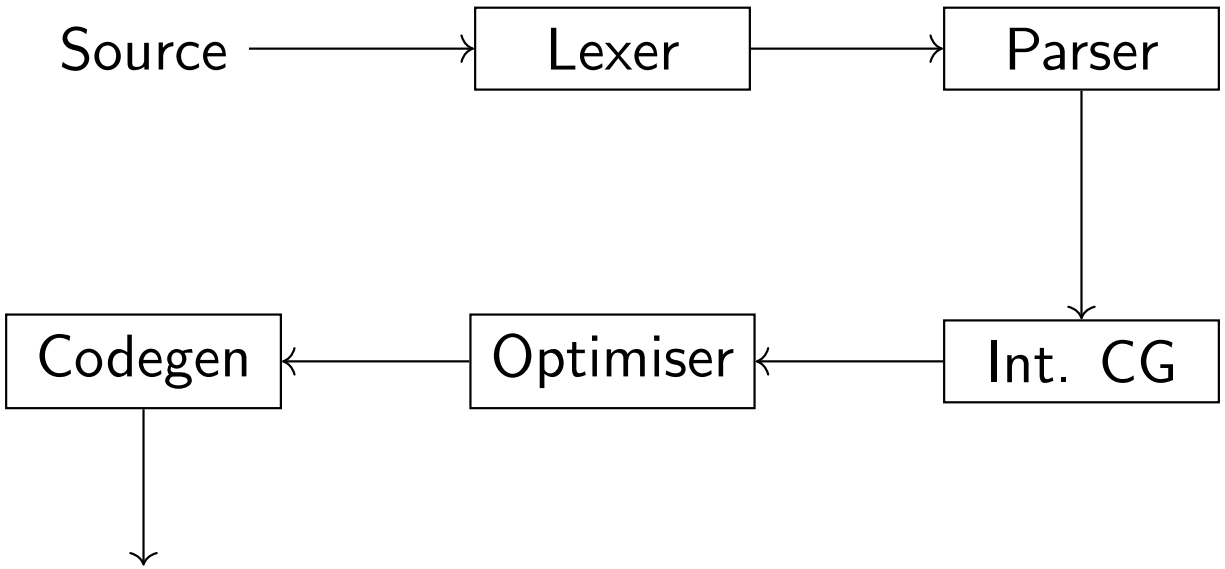
Introduction

The LLVM framework



Introduction

Compilation process



Introduction

LLVM IR

Source A

```
    block := {statement...}
statement := expression;
           | const name = expression;
expression := number | boolean | string
           | name                                  name expression
           | unop expression                       unary operation
           | expression binop expression           binary operation
           | (expression)                          parantheses
   binop := + | - | * | / | % | === | !==
           | > | < | <= | >= | && | ||
    unop := ! | -
```

Source A

Literals

- For a toy language:
  - store and load
- Not enough for dynamic semantics
- Black box of things
  - `double type`
  - Optionally, `double value`, or
  - `int* val`, or
  - `int* Env`, etc.

```
const structType = StructType.create(context, 'literal')
structType.setBody([Type.getDoubleTy(context),
    l.Type.getDoubleTy(context)])

const stringLitType = l.StructType.create(context, 'string_literal')
stringLitType.setBody([l.Type.getDoubleTy(context),
    l.Type.getInt8PtrTy(context)])
```

Source A

Blocks and scoping

- `block` := {`statement` ...}
- Associated with an `Environment`, that contains:
  - `Map<string, Record>` map of name to variable,
  - `Value*` LLVM pointer to the variable,
  - `Environment` reference to parent, etc.

```
const { jumps, offset } = lookupEnv(node.name, env)
const literalStructType = lObj.module.getTypeByName('literal')!
const literalStructPtr = l.PointerType.get(literalStructType, 0)!
const frame = // details skipped

const addr = lObj.builder.createInBoundsGEP(
    literalStructPtr, frame, [
        l.ConstantInt.get(lObj.context, offset)
    ])
lObj.builder.createLoad(addr)
```

Source A

Assignment

- `const name = expression;`
- Evaluate expression,
- wrap result in literal,
- create entries in environment,
- store things in environment

Source A

Binary operations

- `expression1 binop expression2`
- Evaluate `expression1`, evaluate `expression2`
- Type check and branch

```
res = lObj.builder.createFCmpOEQ(leftTypeValue, NUMBER_CODE)
lObj.builder.createCondBr(res, firstNumberBlock, checkFirstString)
lObj.builder.setInsertionPoint(firstNumberBlock)
res = lObj.builder.createFCmpOEQ(rightTypeValue, NUMBER_CODE)
lObj.builder.createCondBr(res, numAddBlock, errorBlock)

lObj.builder.setInsertionPoint(checkFirstString)
res = lObj.builder.createFCmpOEQ(leftTypeValue, STRING_CODE)
lObj.builder.createCondBr(res, checkSecondString, errorBlock)
```

Source B

Conditionals

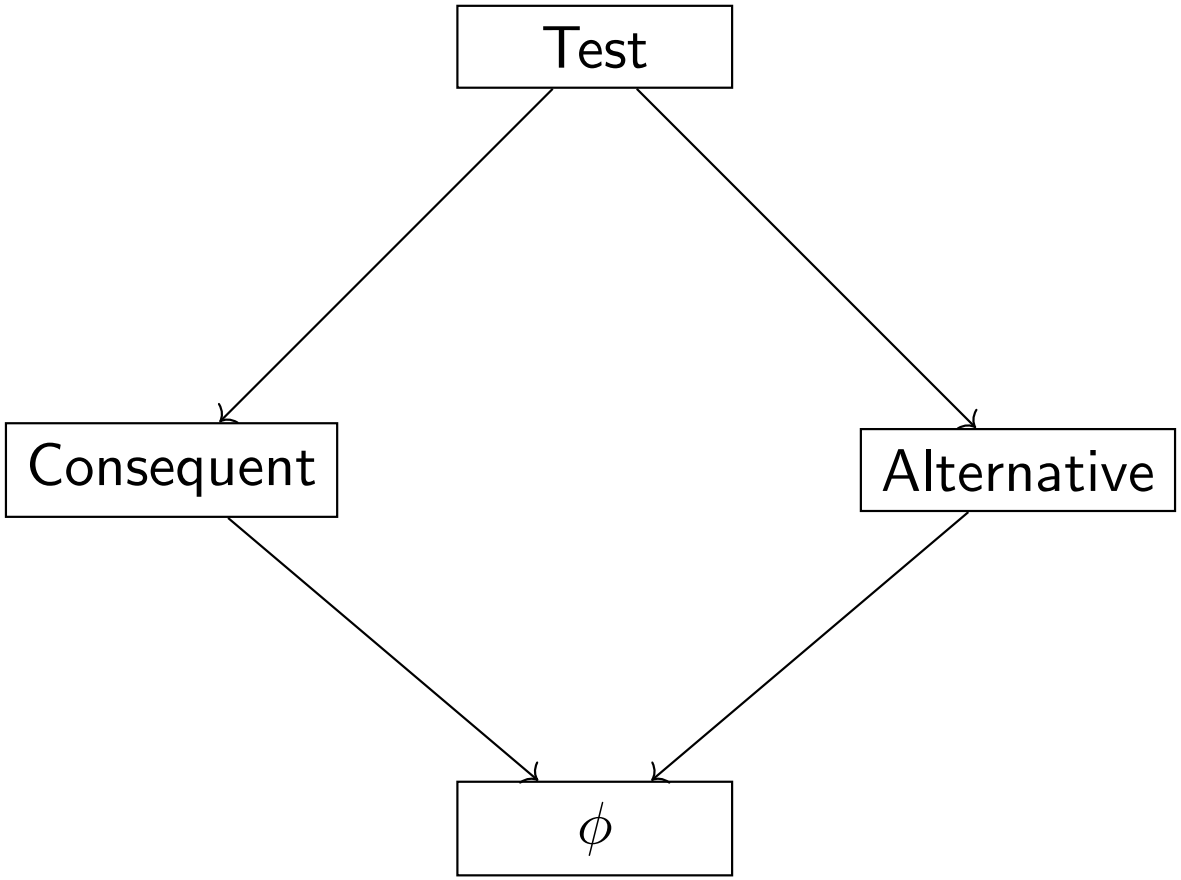


Figure: Control flow

Source B

Conditionals

```
Function evaluate({test_expr, expr1, expr2}, env) {
    consequentBlock = new BasicBlock()
    alternativeBlock = new BasicBlock()
    endBlock = new BasicBlock()

    testResult= evaluate(test_expr, env)
    createBranch(result, consequentBlock , alternativeBlock )

    setInsertionPoint(consequentBlock)
    consequentResult = evaluate(expr1, env)
    createBranch(endBlock)
    // do the same for alternative...

    setInsertionPoint(endBlock)
    phi = createPhi([consequentResult, consequentBlock ],
        [alternativeResult , alternativeBlock])
    return phi
}
```

Source B

Conditionals

$$\frac{\begin{array}{l}\texttt{E1}\rightarrow\texttt{s1, v1}\\ \texttt{E2}\rightarrow\texttt{s2, v2}\\ \texttt{E3}\rightarrow\texttt{s3, v3}\end{array}}{\texttt{E1 ? E2 : E3} \rightarrow \begin{array}{l}\texttt{s1.condbr v1, Con, Alt.}\\ \texttt{Con:. s2. br End.}\\ \texttt{Alt:. s3. br End.}\\ \texttt{end: phi v2, v3}\end{array}}\ [\texttt{? :}]$$

Source C

A problem

- Function frames need to live beyond their lifetime.
- LLVM functions are insufficient to represent Source functions.
- Compare with Java/JLite.

Source C

Grammar

$$
\begin{aligned}
\texttt{expression} &:= \ldots \\
&\mid \texttt{name (params)) => expression} \\
&\mid \texttt{name (params)) => block} \\
\texttt{statement} &:= \ldots \\
&\mid \texttt{function name (params) block} \\
\texttt{params} &:= \epsilon \mid \texttt{name (, name)} \mid \ldots \\
\texttt{expressions} &:= \epsilon \mid \texttt{expression}(, \mathit{expression}) \mid \ldots
\end{aligned}
$$

Source C

Functions

- Function literal:
  - `double type`
  - Function pointer
  - Environment pointer
- Function type:
  - `(*env, *argList) => *function literal`

Source C

Functions

Example

```
function f(x) { return x + 1; }
```

- Prologue:
  - Create a new frame for arguments.
  - Initialize new `Environment` for function.
  - Copy parameters in.
- Evaluate function body.
- Epilogue:
  - Check if block is terminated, otherwise return `undefined`.
- Lookup offset for the name variable and assign the function literal.

Source C

Function calls

$$\frac{\texttt{fun} \vdash \texttt{expression} \rightarrow s1, f \quad \texttt{par} \vdash \texttt{expressions} \rightarrow s1, ..., sn, v1, ..., vn}{\texttt{fun(par)} \rightarrow \texttt{s0,s1,...,sn.} \quad \texttt{call f[env, [v1,...,vn]]}}$$

Source D

Tail calls

- A function is said to be tail recursive if and only if it contains a return expression such that the expression is a tail recursive call.
- An expression is a tail recursive call if and only if it is A call expression that calls the current executing function Or, it is a tenary expression and either the consequent or alternative expression is a tail recursive call.
- Before compiling each function we check if a function is tail recursive and mark those tail recursive calls by prepending "#" to their function names.

Source D

Tail calls

- Evaluate arguments
- Replace arguments in the function's environment
  a: `T1 = evaluate(arg1expr)`
  b: `T2 = evaluate(arg2expr)`
  c: ...
  d: `Env[1] = T1`
  e: `Env[2] = T2`
- Branch to `f`.

Source D

Tail calls

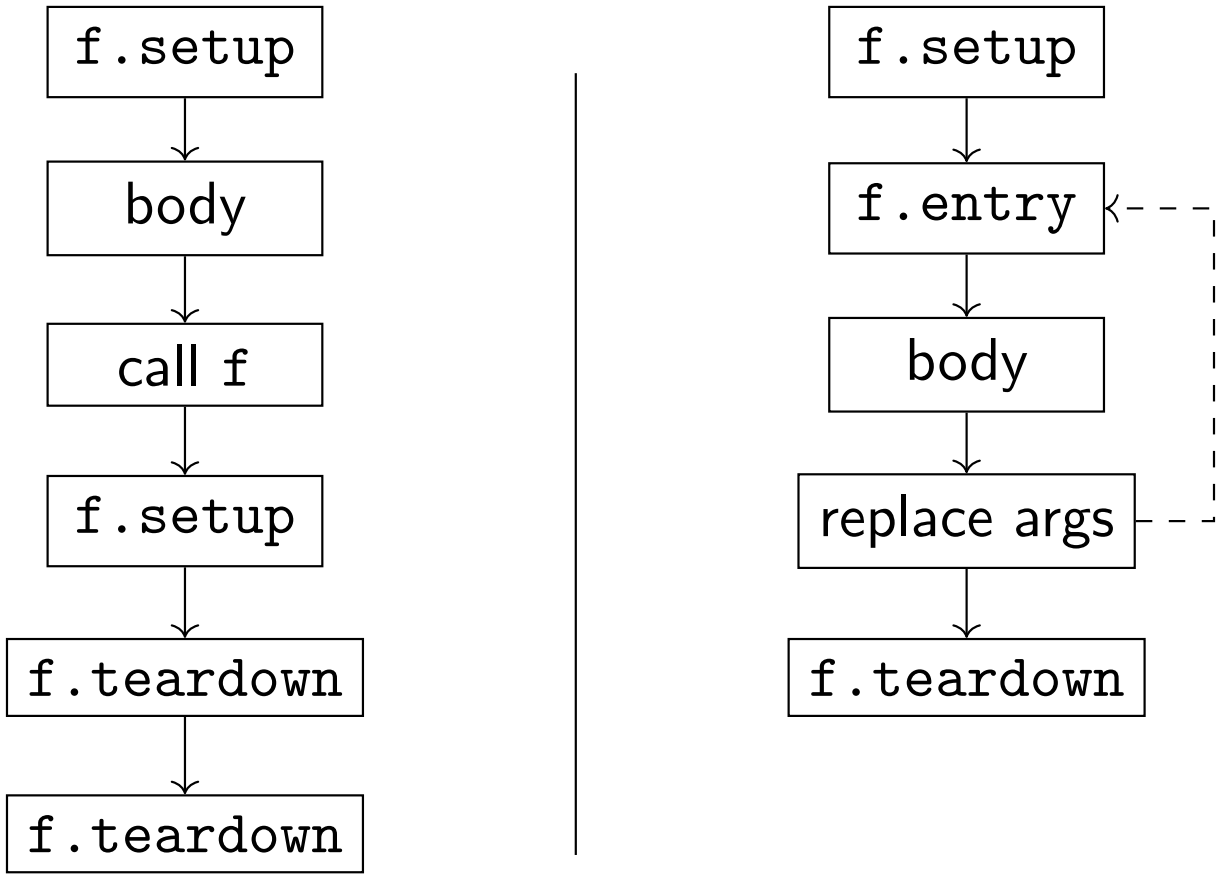


Figure: Tail call execution

Source E

Load store optimisations

- Optimisation 1: check if identifiers are:
- Case 1: they are in the function
  - Use the register with the value
  - Stored in compile time environment
- Case 2: they are live outside the function
  - Use the function environment pointer as usual