IR for Simpler Language Design

Christian Zhao

December 2022

Abstract

A typical compiler consists of a frontend for code parsing and analysis, and a backend for optimizations and code generation. This project provides an intermediate representation (IR) that supports common language features and targets LLVM, in the aim of simplifying compiler design for programming languages.

1 Introduction

Designing a programming languages typically requires implementing an interpreter or a compiler for the language. The source language is first parsed into an intermediate representation (IR), and after some analyses and optimizations on the IR, the compiler then performs code generation to a target platform from the IR or execute the IR directly.

Optimizations and code generation are important for designing an efficient programming language, but implementing separate optimizations and code generation passes for different programming languages is extremely time-consuming. The emergence of LLVM alleviated this issue by providing a unified framework of compiler backend, though generating LLVM code still requires translating high level programming constructs into the LLVM IR. This project aims to provide an IR with more expressive power than LLVM at minimal cost.

The IR of this project allows common constructs of programming languages, for example, while loops with continue and break statements, and closures (nested function definition with access to variables of the outer function). The code generator generates LLVM code directly from the IR. Before code generation, the IR has to be converted into SSA form, which allows some effective optimizations to be done on the IR.

2 The Intermediate Representation

The project employs a tree structured IR that features common basic features of programming languages. The IR nodes consists of two types, statements and expressions. The type system maps directly to the type system of LLVM, which is dynamic and strong.

2.1 Abstract Grammar

The following is the abstract grammar for the IR.

```
Stmt ::= While | Continue | Break
 | Assign | Block | If | Return | Exp

Exp ::= BinOp | StrLiteral | IntLiteral
 | ConvertInt | Var | Fn | Rec | Apply
 | StructArrLiteral | InitArr
 | InitStruct | GetElementAt
 | SetElementAt | Phi | VoidE

# while statement: condition, body, tail
While ::= Exp Block Block
Continue ::= # continue statement
Break ::= # break statement
#  assign value to a variable
Assign ::= string Exp
# block: a list of statements
Block ::= Stmt*
# if statement: condition, if, else
If ::= Exp Block Block
# return statement
Return ::= Exp

# binary integer operation
BinOp ::= Exp Op Exp
# operation code, e.g. ADD - addition
Op ::= ADD | SUB | MUL | DIV | MOD | GT
 | LT | GE | LE | NE | EQ | OR | AND
StrLiteral ::= string
# int value and number of bits
IntLiteral ::= int int
# convert integer expression value
```

```
# to target number of bits
ConvertInt ::= Exp int
# variable name
Var ::= string
#  function expression:
#  return type, parameters, function body
Fn ::= IRType (IRType name)* Block
# recursive function:
# function name and declaration
Rec ::= string Fn
# apply arguments to a function
Apply ::= Exp Exp*
# array initialization:
# array length, element type
InitArr ::= Exp IRType
# struct initialization:
# list of field types
InitStruct ::= IRType+
# a struct or an array literal:
# elements, and whether this is an array
StructArrLiteral ::= Exp* bool
# array/struct access by index:
# array/struct, index
GetElementAt ::= Exp Exp
# insert element to array/struct at an index:
# array/struct, index, element
SetElementAt ::= Exp Exp Exp
# Phi function node:
# variable name and its corresponding block
Phi ::= (Block string)+
VoidE ::= # void expression

# types
IRType ::= IRInt | IRVoid | IRFunction
 | IRArray | IRStruct | Undefined
# integer type: number of bits
IRInt ::= int
IRVoid ::= # void expression
# function type:
# return type and argument types
IRFunction ::= IRType IRType*
# array type:
# element type, optional array size
IRArray ::= IRType [int]
# struct type: field types
IRStruct ::= IRType+
Undefined ::= # placeholder, unknown type
```

In the IR, a program is a `Block`. The IR distinguishes the global scope from local scopes - the global scope is the outermost `Block` statement of the program, whereas a local scope is the `Block` statement of a function body. Global variables, that is, variables assigned by a value outside any function definition, are accessible in the entire program. Local variables, that is, parameters of a function or variables assigned by a value inside a function but not inside any closure of the function, are accessible within the function as well as any closure in the function.

In addition, the IR enforces several rules in order for code generation to work properly: 1. There are no nested Blocks - a `Block` statement cannot directly contain another `Block`. 2. Every variable name needs to be unique, which means that there is no shadowing. 3. Only `Assign` statements where the value expression is constant are allowed in the global scope. A constant expression could be a `Fn`, a `Rec`, `IntLiteral`, `StrLiteral`, or a `StructArrLiteral` containing only `IntLiteral`, `StrLiteral`, and `StructArrLiteral` expressions. 4. The global scope must have a function assignment to a variable named `main`. The function body of `main` is set as the entry point of the program in the code generation phase.

The user can use any of the IR nodes, except for `Phi`, which is used in the SSA form of the IR. The IR has to undergo four transformations before code generation. The first pass performs type check and inference on the IR. The second converts while loops into recursive functions, eliminating `While`, `Break`, and `Continue` statements in the program. The third transformation converts every closure to a global function, eliminating nested functions. The last transformation converts the program into static single assignment (SSA) form, making each variable be assigned by an expression exactly once by adding `Phi` nodes.

2.2 Typing Rules

As the IR targets LLVM directly, it is strongly, statically typed. However, thanks to the type analyzer, there are only a few IR nodes that require types to be given explicitly: `IntLiteral` requires the number of bits of the integer; `ConvertInt` requires the target number of bits; `Fn` requires the return type and the type for each argument; `InitArr` requires the element type; and `InitStruct` requires the type for each field.

The following are the typing rules for the IR.

2.2.1 Expressions

$$\textsc{BinOp}(\text{e1,e2,Op}_1)\ \frac{\Gamma \vdash e_1, e_2 : \mathbf{IRInt(n)}}{\Gamma \vdash e_1\ Op\ e_2 : \mathbf{IRInt(n)}}$$

$\text{Op}_1 \in$ {ADD, SUB, MUL, DIV, MOD, OR, AND}

$$\textsc{BinOp}(\text{e1,e2,Op}_2)\ \frac{\Gamma \vdash e_1, e_2 : \mathbf{IRInt(n)}}{\Gamma \vdash e_1\ Op\ e_2 : \mathbf{IRInt(1)}}$$

$\text{Op}_2 \in$ {GT, LT, GE, LE}

$$\textsc{BinOp}(\text{e1,e2,Op}_3)\ \frac{\Gamma \vdash e_1, e_2 : \mathbf{T}}{\Gamma \vdash e_1\ Op\ e_2 : \mathbf{IRInt(1)}}$$

$\text{Op}_3 \in$ {NE, EQ}

$$\textsc{IntLiteral}(\text{i, n})\ \frac{}{\Gamma \vdash i : \mathbf{IRInt(n)}}$$

$$\textsc{StrLiteral}(\text{s})\ \frac{}{\Gamma \vdash s : \mathbf{IRArray(IRInt(8),\ s.length+1)}}$$

$$\textsc{ConvertInt}(\text{i, n})\ \frac{\Gamma \vdash i : \mathbf{IRInt(m)}}{\Gamma \vdash ConvertInt(i) : \mathbf{IRInt(n)}}$$

$$\textsc{Var}(\text{v})\ \frac{\vdash \langle v : \mathbf{T} \rangle \in \Gamma}{\Gamma \vdash v : \mathbf{T}}$$

$$\textsc{Apply}(\text{f})\ \frac{\Gamma \vdash f : (U_1, \ldots, U_n) \to \mathbf{T} \quad \Gamma \vdash x_1 : U_1\ \ldots\ \Gamma \vdash x_n : U_n}{\Gamma \vdash f(x_1, \ldots, x_n) : \mathbf{T}}$$

$$\textsc{InitArr}(\text{e, }\mathbf{T})\ \frac{\Gamma \vdash e : \mathbf{IRInt(32)}}{\Gamma \vdash InitArr(e, \mathbf{T}) : \mathbf{IRArray(T,e)}}$$

$$\textsc{InitStruct}(\mathbf{T}_1, \ldots, \mathbf{T}_n)\ \frac{}{\Gamma \vdash InitStruct(\mathbf{T}_1, \ldots, \mathbf{T}_n) : \quad \mathbf{IRStruct(T}_1, \ldots, \mathbf{T}_n\mathbf{)}}$$

$$\textsc{StructArrLiteral}(e_1, \ldots, e_n)_{\text{ARRAY}}\ \frac{\Gamma \vdash e_1 : \mathbf{T}\ \ldots\ \Gamma \vdash e_n : \mathbf{T}}{\Gamma \vdash StructArrLiteral(e_1, \ldots, e_n) : \mathbf{IRArray(T,n)}}$$

$$\textsc{StructArrLiteral}(e_1, \ldots, e_n)_{\text{STRUCT}}\ \frac{\Gamma \vdash e_1 : \mathbf{T}_1\ \ldots\ \Gamma \vdash e_n : \mathbf{T}_n}{\Gamma \vdash StructArrLiteral(e_1, \ldots, e_n) : \mathbf{IRStruct(T}_1, \ldots, \mathbf{T}_n\mathbf{)}}$$

$$\textsc{GetElementAt}(\text{arr, i})\ \frac{\Gamma \vdash arr : \mathbf{IRArray(T,n)} \qquad \Gamma \vdash i : \mathbf{IRInt(32)}}{\Gamma \vdash GetElementAt(arr, i) : \mathbf{T}}$$

$$\textsc{GetElementAt}(\text{arr, i})\ \frac{\Gamma \vdash arr : \mathbf{IRStruct(T}_1, \ldots, \mathbf{T}_i, \ldots, \mathbf{T}_n\mathbf{)} \qquad \Gamma \vdash i : \mathbf{IRInt(32)}}{\Gamma \vdash GetElementAt(arr, i) : \mathbf{T}_i}$$

$$\textsc{SetElementAt}(\text{arr, i, e})\ \frac{\Gamma \vdash arr : \mathbf{IRArray(T,n)} \qquad \Gamma \vdash i : \mathbf{IRInt(32)} \qquad \Gamma \vdash e : \mathbf{T}}{\Gamma \vdash SetElementAt(arr, i, e) : \mathbf{IRVoid}}$$

$$\textsc{SetElementAt}(\text{arr, i, e})\ \frac{\Gamma \vdash arr : \mathbf{IRStruct(T}_1, \ldots, \mathbf{T}_i, \ldots, \mathbf{T}_n\mathbf{)} \quad \Gamma \vdash i : \mathbf{IRInt(32)} \quad \Gamma \vdash e : \mathbf{T}_i}{\Gamma \vdash SetElementAt(arr, i, e) : \mathbf{IRVoid}}$$

$$\textsc{Phi}(\phi)\ \frac{\Gamma \vdash \phi : (\mathbf{T}, \ldots, \mathbf{T}) \to \mathbf{T} \quad \Gamma \vdash x_1 : \mathbf{T}\ \ldots\ \Gamma \vdash x_n : \mathbf{T}}{\Gamma \vdash f(x_1, \ldots, x_n) : \mathbf{T}}$$

2.2.2 Statements

$$\textsc{While}\ \frac{\Gamma \vdash e : \mathbf{IRInt(1)}}{\Gamma \vdash while(e)\ b} \qquad \textsc{If}\ \frac{\Gamma \vdash e : \mathbf{IRInt(1)}}{\Gamma \vdash if(e)\ b_1\ else\ b_2}$$

$$\textsc{Assign}(\text{v})\ \frac{\Gamma \vdash e : \mathbf{T}}{v = e \quad \Gamma \vdash add\ \langle v : \mathbf{T} \rangle\ to\ \Gamma}$$

$$\textsc{Return}(\text{from f})\ \frac{\Gamma \vdash f : (U_1, \ldots, U_n) \to \mathbf{T} \quad \Gamma \vdash e : \mathbf{T}}{\Gamma \vdash return\ e}$$

3 IR Transformations

3.1 While Loop Conversion

The while loop conversion eliminates `While`, `Continue`, and `Break` statements by converting while loops into recursive functions.

In the IR, a typical context of a while loop is as following,

```
head
while(condition)
    body
tail
```

where `head` and `tail` are the code before and after the while loop, respectively. The body may contain `Continue` and `Break` statements. We observe that the above code is equivalent to the following code,

```
head
return recursive(env)
```

where `recursive` is defined as the following,

```
def recursive(vars):
    if condition
    then
        tail
    else
        body
        return recursive(vars)
```

and `env` refers to the free variables in `body`, that is, variables that are not assigned in the `body` block.

The conversion is done by iteratively converting each innermost while loop (that is, without any nested while loop) into a corresponding function and a call to the function, and renaming the parameters of the new functions.

3.2 Closure Conversion

The closure conversion eliminates closures by iteratively converting outermost closures to global functions, until there is no closure. In each iteration, the non-global free variables in the closure are packed into a new struct literal as the environment `env`. An argument of the type of `env` is inserted to the closure function definition, and references to the non-global free variables inside the closure are replaced with field accesses to the new `env` struct argument.

The global scope then adds an assignment of the closure function to a new variable, and the closure where it was defined is replaced with a struct of two elements, the first being the new variable that has the closure as value, and the second being the `env` struct literal.

After the above steps are done for all closures, each `Apply` expression in the program that now has the function expression with type `Struct` is transformed as follows: 1. The argument list is appended with the second element of the function expression struct, which is the `env` to be passed, and 2. The function expression is replaced with the first element of the struct, which is the actual `Fn` to be applied to. This transformation fixes the type inconsistency introduced by replacing references to a closure with a struct.

Algorithm 1 gives the pseudocode for closure elimination. The `getNewVar` function returns a variable with a name not used in the program. The `getFreeVars(g)` function returns a list of non-global free variable names in function `g`. The $a \rightarrow b$ operation replaces every reference to a with a reference to b.

Algorithm 1 Closure Elimination

```
1:  procedure closureElim()
2:      for function f in global scope do
3:          closureElim(f)
4:      for apply a in program do
5:          if a.fn.type is IRStruct then
6:              a.args.add(GetElementAt(a.fn,
    1))
7:              a.fn = GetElementAt(a.fn, 0)
8:  procedure closureElim(f)
9:      for closure fn in f do
10:         g ← getNewVar()
11:         freeVars ← getFreeVars(g)
12:         env ← StructLiteral(freeVars)
13:         varPosMap ← HashMap()
14:         pos ← 0
15:         for var in freeVars do
16:             varPosMap.put(var, pos)
17:             pos += 1
18:         newArg ← getNewVar()
19:         fn.addParam(env.type, newArg)
20:         globalBlock.add(Assign(g, fn))
21:         rename(g, env, freeVars, var-
    PosMap)
22:         fn → struct{g, env}
23:         closureElim(fn)
24: procedure rename(g, env, freeVars, var-
    PosMap)
25:     newVars ← freeVars.map(getNewVar)
26:     for variable var in g do
27:         if freeVars.contains(var) then
28:             var → GetElementAt(env, var-
    PosMap.get(var))
```

3.3 SSA Conversion

After while loop conversion and closure elimination, the SSA conversion is performed to convert the IR into SSA form. SSA form improves effectiveness of some optimizations, for example, it makes dead code elimination more accurate than in non-SSA form, and it simplifies code generation from the IR to LLVM, since the LLVM IR is in SSA form.

The conversion uses the algorithm described in Engineering a Compiler [1]. On a high level view, the SSA conversion consists of five steps: 1. CFG construction of the program, 2. find long-lived variables (variables that are live across multiple CFG blocks), 3.compute the dominance frontiers for each CFG block, 4. insert `Phi` expressions to dominance frontiers of each long-lived variable, 5. rename variables so that each variable is assigned exactly once.

Since while loops are converted into functions, the control flow of the program now only branches on `If` statements, and the CFG becomes a directed acyclic graph (DAG). Hence, Construction of CFG is a straightforward traversal of the program. Function calls are not considered in the CFG for the purpose of SSA conversion.

A variable is *long-lived* if it is used in a CFG block in which it is not defined, so

its value is determined at runtime depending on the executed `Block` of `If` statements, and hence require ϕ-functions for SSA conversion. After variable renaming, a variable that is assigned in both if and else branches is renamed to different names, and the ϕ function after the `If` statement determines at runtime which name is assigned a value. Algorithm 2 gives the pseudocode for finding long-lived variables, as well as the set of CFG blocks in which each variable is defined.

Algorithm 2 Find Long-lived Variables

```
1:  procedure findLongLived()
2:      longLivedVars ← ∅
3:      varBlocksMap ← HashMap()
4:      initialize varBlocksMap values to ∅
5:
6:      for each CFG block b do
7:          varKill ← ∅
8:          for each operation i in b, in order do
9:              /* assume op_i is of the form
10:             "x← y op z" */
11:             if y ∉ varKill then then
12:                 longlivedVars.add(y)
13:             if z ∉ varKill then then
14:                 longlivedVars.add(z)
15:             varKill.add(x)
16:             varBlocksMap(x).add(b)
```

A CFG block n *dominates* m if every paths from root to m contains n. The dominator set of m is denoted $DOM(m)$. A CFG block n *strictly dominates* m if $n \in DOM(m)$ and $n \neq m$. The closest dominator of m is denoted as $IDOM(m)$. A *dominator frontier* m of a CFG block n is a joint point (having more than 1 predecessor) in the CFG such that

1. n dominates a predecessor of m.
2. n does not strictly dominate m.

The set of dominator frontiers of a CFG block n is denoted $DF(n)$. A variable assignment in a CFG block n requires a ϕ-function at the beginning of every CFG block in $DF(n)$.

To compute dominator frontiers of each CFG block, we need to construct the *dominator tree*. In a dominator tree, the vertices are the CFG blocks, and two blocks n, m form an edge from n to m if $n = IDOM(m)$.

A depth first search on the CFG from root is performed to build the dominator tree. We store the ancestors for each block. At each block, if the next child block to traverse has some ancestors recorded, we keep only the shared ancestors between the block and the child block. The traversal must terminate, as the CFG is a DAG.

After constructing the dominator tree, we are able to compute the dominator frontiers. Algorithm 3 gives the pseudocod. There are three important observations that are applied in the algorithm:

1. Only joint points can be contained in a dominator frontier set
2. A joint point is a dominator frontier of each of its predecessor
3. If $j \in DF(k)$, $l \in DOM(k)$, and $l \notin DOM(j)$, then $j \in DF(j)$

Algorithm 3 Compute Dominator Frontiers

```
1:  procedure compute()
2:      for each CFG block b do
3:          DF(b) ← ∅
4:      for each CFG block b do
5:          if b has > 1 predecessors then
6:              for each predecessor p of b do
7:                  runner ← p
8:                  while runner ≠ IDOM(b) do
9:                      DF(runner).add(b)
10:                     runner ← IDOM(runner)
```

We can then add the ϕ-functions to every join point on the path started from the definition of x, as shown in Algorithm 4.

Algorithm 4 Insert Phi Functions

```
1:  procedure insertPhi()
2:      for each name x in longLivedVars do
3:          worklist ← varBlocksMap(x)
4:          for each block b in worklist do
5:              for each block d in DF(b) do
6:                  if d has no φ for x then
7:                      insert φ for x in d
8:                      worklist.add(d)
```

The last step, renaming, ensures that each variable name is assigned with a value exactly once. We keep a counter and a stack for each variable v. On each assignment to v, we increase the counter and add the new name to the stack. At the end of a basic

Algorithm 5 Rename Variables

```
1: procedure RENAME()
2:     for each name i ∈ longLivedVars do
3:         counter[i] ← 0
4:         stack[i] ← ∅
5:     RENAME(root)
6: procedure RENAME(b)
7:     for each φ in b, "x ← φ(···)" do
8:         rewrite x as GETNEWNAME(x)
9:     for each operation "x ← y op z" in b do
10:        rewrite y with subscript top(stack[y])
11:        rewrite z with subscript top(stack[z])
12:        rewrite x as GETNEWNAME(x)
13:    for each successor of b do
14:        fill in φ-function parameters
15:    for each successor s of b in the dominator
   tree do
16:        RENAME(s)
17:    for each operation "x ← y op z" in b
18: and each φ "x ← φ(···)" do
19:        pop(stack[x])
20: procedure GETNEWNAME(()n)
```

block, pop all new names from stacks in this block. Algorithm 5 shows the pseudocode for renaming.

4 Code Generation

Most of the IR nodes have a straight forward translation to LLVM, except for a few exceptions. `InitArr`, `InitStruct`, and `StructArrLiteral` are used to create fixed-size arrays (values of the same type placed on a contiguous memory) and structs ("ragged" array in which each element can have a different type). At the moment, except for `StructArrLiteral` assignment in global scope, in which case the struct/array uses the static area, arrays and structs can be created only on heap (using a call to `malloc`), and cannot be garbage collected.

5 Future Work

There are several directions for future work on this project. The IR currently lacks the ability to free up memory used by arrays and structs, and one could add another pass to the IR to make automatic garbage collection, or create API for the language designer to implement custom garbage collection mechanism. The IR could also be expanded to support more advanced language features, such as exceptions and dynamic typing, which requires a conventional implementation for the data wrapper. Partial evaluation and optimizations on the IR can be implemented on any stage of the IR, and have the potential to significantly improve the performance of the generated code. After partial evaluation is implemented, one could write a parser of Scala (the language in which the project is implemented) that targets our IR, and experiment the Futamura projections [2] to obtain a program that takes an interpreter as input and outputs a compiler.

References

[1] Cooper, Keith D. & Torczon, Linda (2003). Engineering a Compiler. Morgan Kaufmann. ISBN 978-1-55860-698-2.

[2] Futamura, Y. (1983). Partial computation of programs. In: Goto, E., Furukawa, K., Nakajima, R., Nakata, I., Yonezawa, A. (eds) RIMS Symposia on Software Science and Engineering. Lecture Notes in Computer Science, vol 147. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-11980-9_13

Appendix

Given a program P, we say that its static input s is the data of the program known at compile time, and its dynamic input d is the data of the program given at runtime.

The partial evaluator (or specializer) E takes a program P as input, tries to compute part of the program at compile time based on its static input s, and outputs the specialized program called the residual program, P_s.

Denoting $P(s, d)$ as the output of executing P with input (s, d) and $P_s(d)$ as the output of executing P_s with input d, we have that $P(s, d) = P_s(d)$.

Futamura Projections

1. Consider an interpreter I that takes a program P and the input of the program s, d as input, and executes it. We have that $I(P, (s, d)) = P(s, d)$.

   Given an interpreter I and a program P, we can specialize I on P, that is, $E(I, P) = I_P$. Since $I_P(s, d) = I(P, (s, d)) = P(s, d)$ We notice that I_P is an executable of P. This is known as the first Futamura projection. At this point we wouldn't say E is a compiler because it is too slow.

2. A compiler C takes a program P as input and outputs an executable of P. Thus, $C(P)(s, d) = I(P, (s, d))$.

   Now, consider specializing E itself on the interpreter I. We have that

   $$\begin{aligned} E(E, I)(P)(s, d) &= E_I(P)(s, d) \\ &= I_P(s, d) \\ &= P(s, d). \end{aligned}$$

   Thus, we have obtained a compiler E_I. This is known as the second Futamura projection.

3. Last, consider specializing E itself on itself. Then,

   $$E(E, E)(I) = E_E(I) = E(E, I) = E_I$$

   We have obtained a program E_E that is capable of producing a compiler for any interpreter - a compiler generator (*cogen*). Specializing the specializer on itself is known as the third Futamura projection.

Implementation Considerations of Futamura Projections

Specializing the specializer itself requires it to be implemented in the same language as its target language. For most languages this is impractical since implementing effective partial evaluation directly on them is hard. One possibility is to design a IR suitable for partial evaluation, and then write a compiler that is able to convert the specializer code to the IR (so that the specializer can be fed as input to itself) and then convert the IR to a low-level language (so that the cogen is an executable).

One might also consider applying the bootstrapping method as in implementing compilers to gradually write the specializer in this IR, but writing code direct in IR is hardly feasible from the engineering perspective.