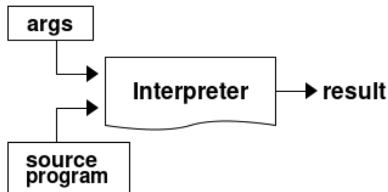
LLVM.MIX — MULTI-STAGE COMPILER-ASSISTED SPECIALIZER GENERATOR BUILT ON LLVM

Eugene Sharygin[1]

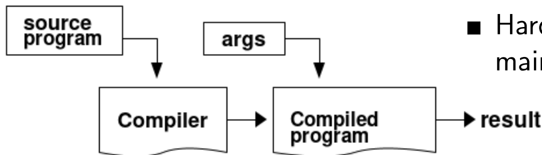
February 3, 2019

[1] eush77@gmail.com

Interpreters and Compilers

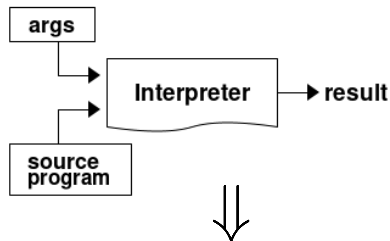


- Direct translation of language semantics
- Easy to understand, debug, extend, and verify
- Interpretation overhead



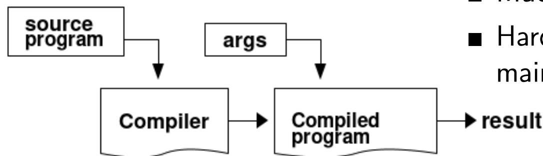
- Multi-stage execution
- Much better performance
- Hard to develop and maintain

INTERPRETERS AND COMPILERS



- Direct translation of language semantics
- Easy to understand, debug, extend, and verify
- Interpretation overhead

⇓

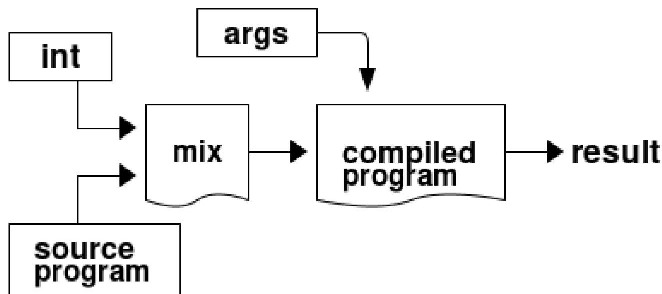


- Multi-stage execution
- Much better performance
- Hard to develop and maintain

Partial Evaluation

$$\|mix\|(p, x) = p_1$$
$$\|p_1\|(y) = \|p\|(x, y)$$

Given $p =$ int, $x =$ source program, $y =$ args:



Separation of Binding Times

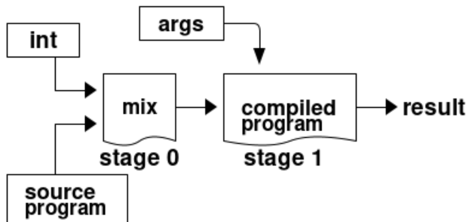
In order to apply partial evaluation, we need separated binding times:

interpreter(source program , args)

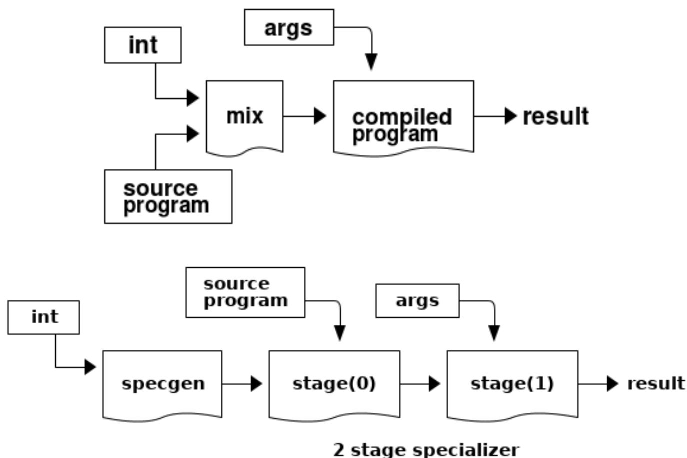
Binding Times:

- source program — stage 0 (*static*)
- args — stage 1 (*dynamic*)

An argument with binding-time stage N is fixed from stage N onward.

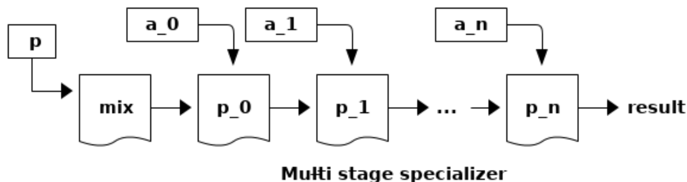


From Partial Evaluation to Specializer Generation



Multi-Stage Specializer Generation

$$program(a_0, a_1, a_2, \ldots, a_n)$$
$$stage(a_k) = k$$



Multi stage specializer

E. g. for query processing:

- Configuration options (stage 0)
- Prepared statements and stored procedures (stage 1)
- Query parameters (stage 2)
- Stored data (stage 3)

Multi-Stage Specializer Generation for LLVM

- Language-agnostic algorithm → language-independent optimizer
- Enabling lots of languages:
  - C, C++, ObjC, etc
  - Fortran
  - Julia
  - Rust
  - Swift
  - ...

LLVM/Clang Extensions

Attributes:

- `stage(k)` (LLVM: functions, returns, parameters)
- `__stage(k)` (Clang: functions, returns, parameters, struct fields)
- `__attribute__((mix(f)))` (Clang: functions)
- `__attribute__((staged))` (Clang: structs)

Intrinsics:

- `declare i8* @llvm.mix(i8*, i8*, ...)`
- `declare i8* @llvm.mix.call(i8*, ...)`
- `declare i32* @llvm.object.stage.p0i32(i32*, i32)`

Built-ins:

- `void *__builtin_mix_call(void *, ...)`

Passes:

- `llvm/lib/Transforms/Mix/Mix.cpp`
- `llvm/lib/Analysis/BindingTimeAnalysis.cpp`

Binding Time Annotations

- stage is a function/return/parameter attribute
- stage(0) is the default

LLVM IR

```
declare stage(1) i32 @add(stage(1) i32 %x, i32 %y) stage(1)
```

C

```
__stage(1) int add(__stage(1) int X, int Y) __stage(1) {
  return X + Y;
}
```

Interface

Specializer interface in LLVM

```
@llvm.mix(@add, i32 4)   ; -> stage(1) specializer
```

Specializer generator interface in Clang

```
__attribute__((mix(add)))
Function *mixAdd(LLVMContext *, int);

// ...
mixAdd(Ctx, 4)  // -> stage(1) specializer
```

Usage

Defining a mix function

```
__stage(1) int add(__stage(1) int X, int Y) __stage(1) {
  return X + Y;
}

__attribute__((mix(add)))
Function *mixAdd(LLVMContext *Ctx, int Y);
```

Compiling with Orc

```
auto Ctx = std::make_unique<LLVMContext>();

Function *F = mixAdd(&Ctx, 1) ;
JIT.addIRModule(ES.getMainJITDylib(),
       ThreadSafeModule(std::unique_ptr<Module>(F->getParent()), Ctx));
auto *Inc = reinterpret_cast<int (*)(int)>(
       ES.lookup({&ES.getMainJITDylib()}, ES.intern(F->getName()))
         ->getAddress());

Inc(4);
//=> 5
```

Clang CodeGen

```
__attribute__((mix(add))) Function *mixAdd(LLVMContext *Ctx, int Y);
```

⇓Clang

```
define %struct.Function* @mixAdd(%struct.LLVMContext* %Ctx, i32 %Y) {
  %Ctx13 = bitcast %struct.LLVMContext* %Ctx to i8*
  %function = call i8* (i8*, i8*, ...) @llvm.mix(
          i8* bitcast (i32 (i32, i32)* @add to i8*),
          i8* %Ctx13,
          i32 %Y)
  %function4 = bitcast i8* %function to %struct.Function*
  ret %struct.Function* %function4
}

declare i8* @llvm.mix(i8*, i8*, ...)
```

Mix Transformation

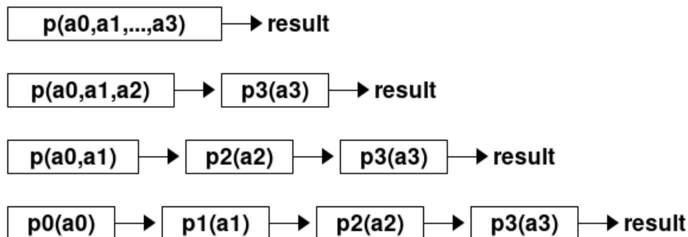
```
define %struct.Function* @mixAdd(%struct.LLVMContext* %Ctx, i32 %Y) {
  %Ctx13 = bitcast %struct.LLVMContext* %Ctx to i8*
  %function = call i8* (i8*, i8*, ...) @llvm.mix(
      i8* bitcast (i32 (i32, i32)* @add to i8*), i8* %Ctx13, i32 %Y)
  %function4 = bitcast i8* %function to %struct.Function*
  ret %struct.Function* %function4
}
```

$\Downarrow$Mix

```
define %struct.Function* @mixAdd(%struct.LLVMContext* %Ctx, i32 %Y) {
  %Ctx131 = bitcast %struct.LLVMContext* %Ctx
                to %struct.LLVMOpaqueContext*
  %function2 = call %struct.LLVMOpaqueValue* @add.main(
      %struct.LLVMOpaqueContext* %Ctx131, i32 %Y)
  %function4 = bitcast %struct.LLVMOpaqueValue* %function2
                   to %struct.Function*
  ret %struct.Function* %function4
}
```

Function Staging

- For functions with $N+1$ binding times, apply the staging transformation N times:



Function Staging (IR)

```
declare stage(2) i32 @F(i32 %x,
                        stage(1) i32 %y,
                        stage(2) i32 %z) stage(2)
; ⇓
declare stage(1) %struct.LLVMOpaqueValue* @G(
                    stage(1) i8** %mix.context,
                    i32 %x,
                    stage(1) i32 %y) stage(1)
```

- `@G` evaluates operations of stages 0, 1 and creates code to evaluate operations of stage 2.
- Argument `%z` is moved to the residual function.
- `@G` loads `LLVMContext`, `Module`, `IRBuilder`, etc from `%mix.context` argument.

Function Staging (One More Step)

```
declare stage(2) i32 @F(i32 %x,
                        stage(1) i32 %y,
                        stage(2) i32 %z) stage(2)

; ⇓
declare stage(1) %struct.LLVMOpaqueValue* @G(
                     stage(1) i8** %mix.context,
                     i32 %x,
                     stage(1) i32 %y) stage(1)

; ⇓
declare %struct.LLVMOpaqueValue* @H(i8** %mix.context, i32 %x)
```

Basic Block Example

Basic blocks of $stage < N$	$\rightarrow$	Basic blocks
Basic blocks of $stage = N$	$\rightarrow$	LLVMAppendBasicBlock
Instructions of $stage < N$	$\rightarrow$	Instructions
Instructions of $stage = N$	$\rightarrow$	LLVMBuildInstr

```
A:              ; stage(0)
  br i1 %b, label %B,
            label %C
                ; stage(1)

B:              ; stage(1)
  %r0 = add i32 %x, 1
                ; stage(1)
  br label %C   ; stage(1)

C:              ; stage(1)
  %r1 = add i32 %y, 1
                ; stage(0)
  br label %D   ; stage(0)

D:              ; stage(0)
```

$\overset{}{\underset{N=1}{\Longrightarrow}}$

```
A:
  %7 = call %struct.BasicBlock* @LLVMAppendBasicBlock
  call void @LLVMPositionBuilderAtEnd
  %B = call %struct.BasicBlock* @LLVMAppendBasicBlock
  %C = call %struct.BasicBlock* @LLVMAppendBasicBlock
  %8 = call %struct.Value* @LLVMBuildCondBr
  call void @LLVMPositionBuilderAtEnd
  %9 = call %struct.Value* @LLVMConstInt
  %r0 = call %struct.Value* @LLVMBuildBinOp
  %10 = call %struct.Value* @LLVMBuildBr
  call void @LLVMPositionBuilderAtEnd
  %r1 = add i32 %y, 1
  br label %D

D:
```

Calls

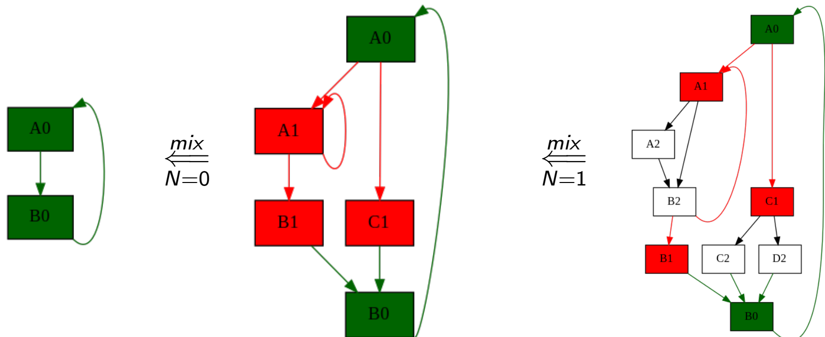
- All calls to other staged functions are split into two:

```
declare stage(1) i32 @f(i32 %x, i32 stage(1) %y) stage(1)

; ...
%call = call i32 @f(i32 %x, i32 %y)
```

```
%f1 = call %struct.LLVMOpaqueValue* @f.mix(i8** %mix.context, i32 %x)
%args = alloca %struct.LLVMOpaqueValue*
store %struct.LLVMOpaqueValue* %y, %struct.LLVMOpaqueValue** %args
%call10 = call %struct.LLVMOpaqueValue* @LLVMBuildCall(
                   %struct.LLVMOpaqueBuilder* %builder7,
                   %struct.LLVMOpaqueValue* %f1,
                   %struct.LLVMOpaqueValue** %args,
                   i32 1,
                   i8* @mix.name)
```

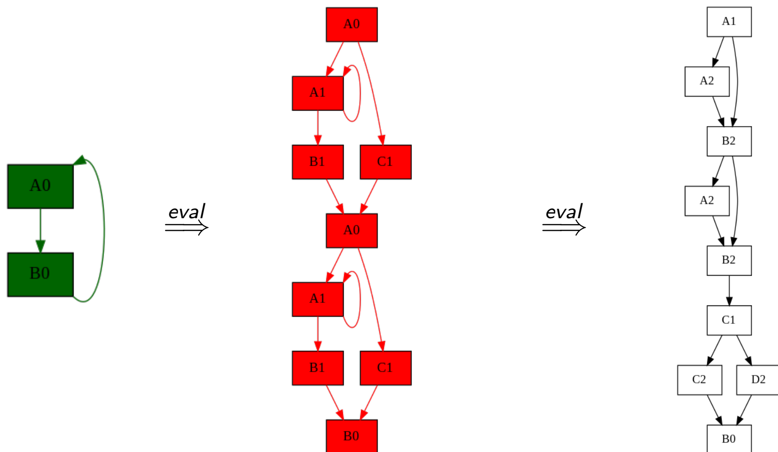
Control Flow Folding

- Mix iteratively folds the control flow graph down to stage 0:
  1. Pick a branch connecting a basic block of stage $\leq N$ with a basic block B of stage $> N$
  2. Replace the branch with branches to successors of B
  3. Repeat 1–2 until there are no such branches left
  4. Remove unreachable blocks



Control Flow Folding (Execution)

- When evaluated, each intermediate stage recreates the next stage using pieces of the original control flow graph:



Binding-Time Analysis

$$stage(Instr)$$

$$stage(BB)$$

- Minimum fixed-point algorithm
- Last-stage operations:
  - Calls to external functions
  - `alloca`, non-annotated memory operations
  - Operations with type `void` or unmodelled side effects
- Any **contradiction** is diagnosed as an error
- Ambiguities are resolved arbitrarily but also reported

Analysis Results

tmp.c

```
__stage(2) int f(int x, __stage(1) int y, __stage(2) int z) __stage(2){
  return (x / 2 < x - y) ? x - y : y / z;
}
```

```
$ clang -S -emit-llvm -O1 tmp.c -o - | opt -analyze -bta
```

```
Printing analysis 'Binding-Time Analysis' for function 'f':

define stage(2) i32 @f(i32 %x, i32 stage(1) %y, i32 stage(2) %z) stage(2) {
entry:
; stage(0)
  %div = sdiv i32 %x, 2                          ; stage(0)
  %sub = sub nsw i32 %x, %y                      ; stage(1)
  %cmp = icmp slt i32 %div, %sub                 ; stage(1)
  br i1 %cmp, label %cond.end, label %cond.false  ; stage(1)

cond.false:
; stage(1)
  %div2 = sdiv i32 %y, %z                        ; stage(2)
  br label %cond.end                             ; stage(1)

cond.end:
; stage(1)
  %cond = phi i32 [ %div2, %cond.false ], [ %sub, %entry ] ; stage(1)
  ret i32 %cond                                  ; stage(2)
}
```

Binding-Time Rules

$$\frac{a \text{ has } \texttt{stage(N)} \text{ attr}}{stage(a) = N} \ (\textsc{Param})$$

$$\frac{op \neq \varphi}{stage(op(\ldots, a, \ldots)) \geq opstage(a)} \ (\textsc{Operand})$$

$$\frac{B' \in succ(B)}{stage(term(B)) = stage(B')} \ (\textsc{BasicBlock})$$

$$\frac{\varphi \in \Phi(B)}{stage(\varphi) = stage(B)} \ (\textsc{Phi})$$

$$\frac{\begin{array}{c} ret(\ldots) \in reach_N(B) \\ T \in reach_N(B) \end{array}}{stage(T) > N} \ (\textsc{Ret})$$

$$\frac{\begin{array}{c} term(B') \in reach_N(B) \\ call(f, \ldots) \in B' \\ f \text{ is a staged func} \end{array}}{B' \in postdom(B)} \ (\textsc{StaticCall})$$

$$\frac{\begin{array}{cc} T \in reach_N(B) & stage(T) \leq N \\ T' \in reach_N(B) & stage(T') \leq N \end{array}}{T = T'} \ (\textsc{SingleTerm})$$

$$\frac{objectstage(p) = N}{stage(load(p)) \geq N} \ (\textsc{Load})$$

$$\frac{objectstage(p) = N}{stage(store(x, p)) = N - 1} \ (\textsc{Store})$$

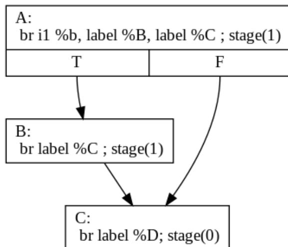
Terminators

- Every basic block must have at most one reachable terminator at each stage:

$$term(B) \in reach_N(B) \qquad \frac{stage(term(B)) > N \qquad B' \in succ(B)}{reach_N(B') \subseteq term_N(B)}$$

$$\frac{\begin{matrix} T \in reach_N(B) & stage(T) \leq N \\ T' \in reach_N(B) & stage(T') \leq N \end{matrix}}{T = T'} \ (\textsc{SingleTerm})$$

- Reachable terminators of stage N become terminators of basic blocks of the specializer at stage N:



$\Rightarrow$

```
A:
 call void @LLVMPositionBuilderAtEnd
 %8 = call %struct.Value* @LLVMBuildCondBr
 call void @LLVMPositionBuilderAtEnd
 %9 = call %struct.Value* @LLVMBuildBr
 call void @LLVMPositionBuilderAtEnd
 br label %D
```

Termination

Theorem (Termination of Specializer)

All intermediate execution stages terminate if the source program terminates on the same input.

Dynamic Control

- Some forms of mixing static and dynamic computation are very common:

```
__stage(1) int eval(Node *) __stage(1);

int f() __stage(1) {
  // ...
  for (Node *N = Nodes.begin; N != Nodes.end; ++N) {
    int Val = eval(N);          // BTA error: expected stage(0) argument

    if (Val)                                                // dynamic control
      break;
  }
}
```

- Although the complete set of `Nodes` is known at stage 0, `eval` can't be specialized because `stage(N)=1` in the loop.

Mix Call

```
declare stage(1) i8* @llvm.mix.call(i8*, ...)
```

```
__stage(1) void *__builtin_mix_call(void *, ...)
```

- Mix call builtin/intrinsic calls the stage 0 of the function with `stage(0)` arguments and returns a residual function pointer that continues the computation.

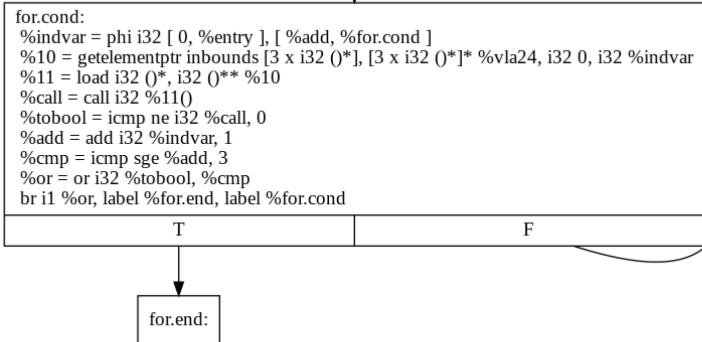
```
__stage(1) int eval(Node *) __stage(1);

int f() __stage(1) {
  // ...
  int (*Funcs[Nodes.end - Nodes.begin])(void);
  for (Node *N = Nodes.begin; N != Nodes.end; ++N)          // static
    Funcs[N - Nodes.begin] = __builtin_mix_call(eval, N);

  for (Node *N = Nodes.begin; N != Nodes.end; ++N) {        // dynamic
    int Val = Funcs[N - Nodes.begin]();

    if (Val)
      break;
  }
}
```

Mix Call

```
__stage(1) int eval(Node *) __stage(1);

int f() __stage(1) {
  // ...
  int (*Funcs[Nodes.end - Nodes.begin])(void);
  for (Node *N = Nodes.begin; N != Nodes.end; ++N)             // static
    Funcs[N - Nodes.begin] = __builtin_mix_call(eval, N);

  for (Node *N = Nodes.begin; N != Nodes.end; ++N) {          // dynamic
    int Val = Funcs[N - Nodes.begin]();

    if (Val)
      break;
  }
}
```

- This helps separate computation at different stages.
- Functions using `__builtin_mix_call` cannot be codegened by the normal pipeline and can only be used with Mix.

Residual CFG



Arithmetic Expressions

```
__stage(1) int
eval(Node *E, __stage(1) int *Args)
__stage(1) {
  BinaryNode *BE = (BinaryNode *)E;
  switch (E->Op) {
  case O_Int:
    return ((Int *)E)->Value;
  case O_Param:
    return Args[((Param *)E)->Number];
  case O_Add:
    return eval(BE->Left, Args) +
           eval(BE->Right, Args);
  case O_Mul:
    return eval(BE->Left, Args) *
           eval(BE->Right, Args);
  case O_Div:
    return eval(BE->Left, Args) /
           eval(BE->Right, Args);
  }
}
```

⇒

```
; E = (Div (Add (Add (Param 0) (Int 1))
;                    (Mul (Param 0)
;                         (Param 1)))
;               (Int 2))
define i32 @eval(i32* %Args) {
  %0 = load i32, i32* %Args
  %add.i.i = add i32 %0, 1
  %arrayidx.i.i.i = getelementptr
      inbounds i32, i32* %Args, i64 1
  %1 = load i32, i32* %arrayidx.i.i.i
  %mul.i.i = mul i32 %1, %0
  %add.i = add i32 %add.i.i, %mul.i.i
  %div = sdiv i32 %add.i, 2
  ret i32 %div
}
```

Matrix Convolution

```
for (Row = KH/2; Row < H - KH/2;
     ++Row) {
  for (Col = KW/2; Col < W - KW/2;
       ++Col) {
    double S = 0;
    for (KRow = 0; KRow < KH;
         ++KRow) {
      for (KCol = 0; KCol < KW;
           ++KCol)
        S += Kernel[
           KW * KRow + KCol] *
          Image[
            W * (Row - KH/2 + KRow)
            + (Col - KW/2 + KCol)];
    }
    Out[W * Row + Col] = S;
  }
}
```

⇒

$$; \; Kernel = \begin{matrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{matrix}$$

```
%add.i = fsub double 0.000000e+00, %1
%add6.i = fsub double %add.i, %2
%add12.i = fsub double %add6.i, %3
%add23.i = fsub double %add12.i, %4
%mul28.i = fmul double %5, 9.000000e+00
%add29.i = fadd double %add23.i, %mul28.i
%add35.i = fsub double %add29.i, %6
%add47.i = fsub double %add35.i, %7
%add53.i = fsub double %add47.i, %8
%add59.i = fsub double %add53.i, %9
store double %add59.i, double* %arrayidx
```

String Formatting

```
void format(__stage(1) char *Out, const char *Fmt,
            __stage(1) const char **Args) __stage(1) {
  do {
    const char *Subst = Fmt;

    while (Subst[0] &&
           (Subst[0] != '{'
            Subst[1] < '0'
            Subst[1] > '9'
            Subst[2] != '}'))
      Subst += 1;

    if (Subst != Fmt) {
      memcpy(Out, Fmt, Subst - Fmt);
      Out += Subst - Fmt;
    }

    if (*Subst) {
      unsigned NSub = Subst[1] - '0';
      size_t Len = strlen(Args[NSub]);
      memcpy(Out, Args[NSub], Len);
      Out += Len;
      Fmt = Subst + 3;
    } else {
      Fmt = Subst;
    }
  } while (*Fmt);
}
```

```
; Fmt = "Good morning, {0} {1}, and welcome to..."
```

```
call @llvm.memcpy(%Out, 93929996318656, 14)
%ptr31 = getelementptr i8, %Out, 14
%0 = load i8*, %Args
%call = call i64 @strlen(%0)
call @llvm.memcpy(%ptr31, %0, %call)
%ptr43 = getelementptr i8, %ptr31, %call
%1 = load i8, 93929996318673   ;=> 32
store i8 %1, %ptr43
%ptr3137 = getelementptr i8, %ptr43, 1
%arrayidx4040 = getelementptr i8*, %Args, 1
%2 = load i8*, %arrayidx4040
%call41 =  call i64 @strlen(%2)
call @llvm.memcpy(%ptr3137, %2, %call41)
%ptr4342 = getelementptr i8, %ptr3137, %call41
call @llvm.memcpy(%ptr4342, 93929996318677, 19)
```

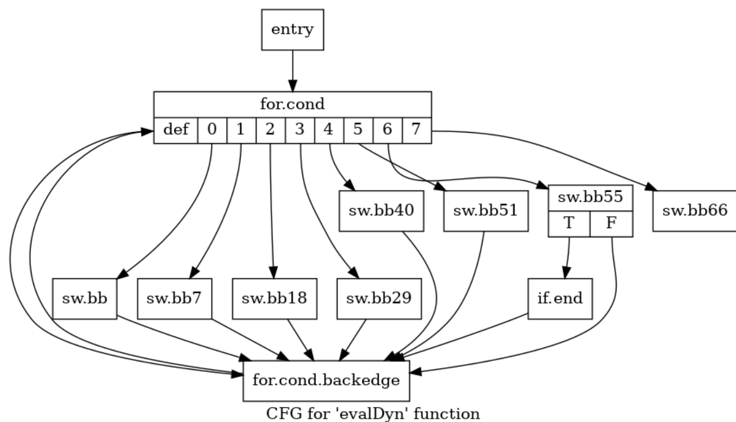
Best Cases Summary

```
------------------------------------------------------------------------
Benchmark                              Time             CPU   Iterations
------------------------------------------------------------------------
BM_Arith                            20.5 ns         20.5 ns     33749103
BM_ArithMix                         1.45 ns         1.45 ns    483128097
BM_Convolution                     94780 ns        94750 ns         7343
BM_ConvolutionMix                  24219 ns        24210 ns        28763
BM_Format                           42.6 ns         42.6 ns     16553156
BM_FormatMix                        11.1 ns         11.1 ns     63807687
```

Bytecode Interpreter

```
__stage(1) int eval(struct Instruction *PC,
                    __stage(1) int *Args) __stage(1) {
  for (;;) {
    switch (PC->Op) {
    case O_Int: Regs[PC->Operands[0]] = PC->Operands[1];
                break;
    case O_Par: Regs[PC->Operands[0]] = Args[PC->Operands[1]];
                break;
    case O_Add: Regs[PC->Operands[0]] += Regs[PC->Operands[1]];
                break;
    case O_Jmp: PC += PC->Operands[0];
                continue;
    case O_Jze: if (Regs[PC->Operands[0]])          // dynamic control
                  break;
                PC += PC->Operands[1];
                continue;
    case O_Ret: return Regs[PC->Operands[0]];
    }
    PC += 1;
  }
}
```

Bytecode Interpreter (cont.)



CFG for 'evalDyn' function

Bytecode Interpreter (cont.)

```
------------------------------------------------------------------
Benchmark                              Time           CPU Iterations
------------------------------------------------------------------
BM_Bytecode/Dyn                     87.0 ns       87.0 ns    7976372
BM_BytecodeMix/Dyn                  78.6 ns       78.6 ns    8680200
```

Bytecode Interpreter: Second Try

```
___stage(1) int evalInstruction(struct Instruction *PC,
                                 __stage(1) int *Args) __stage(1) {
  switch (PC->Op) {
  case O_Int: Regs[PC->Operands[0]] = PC->Operands[1];
              return 1;
  case O_Par: Regs[PC->Operands[0]] = Args[PC->Operands[1]];
              return 1;
  case O_Add: Regs[PC->Operands[0]] += Regs[PC->Operands[1]];
              return 1;
  case O_Jmp: return PC->Operands[0];
  case O_Jze: return Regs[PC->Operands[0]] ? PC->Operands[1] : 1;
  case O_Ret: return 0;
  }
}
int eval(unsigned ProgramSize, struct Instruction Program[ProgramSize],
         int *Args) {
  int PC = 0, Delta;
  while ((Delta = evalInstruction(&Program[PC], Args)))
    PC += Delta;                                    // dynamic control
  return Regs[Program[PC].Operands[0]];
}
```

Bytecode Interpreter: Second Try (cont.)

- Need a separate *driver function* for Mix:

```
__stage(1) int evalForMix(unsigned ProgramSize,
                          struct Instruction *Program,
                          __stage(1) int *Args) __stage(1) {
  int (*Funcs[ProgramSize])(int *);

  for (struct Instruction *I = Program; I != Program + ProgramSize; ++I)
    Funcs[I - Program] = __builtin_mix_call(evalInstruction, I);

  int PC = 0, Delta;
  while ((Delta = Funcs[PC](Args)))
    PC += Delta;
  return Regs[Program[PC].Operands[0]];
}

__attribute__((mix(evalForMix))) void *mix(void *, unsigned, Instruction *);
```

Bytecode Interpreter: Second Try (cont.)

```
---------------------------------------------------------------------
Benchmark                              Time             CPU   Iterations
---------------------------------------------------------------------
BM_Bytecode/Dyn                     87.0 ns         87.0 ns      7976372
BM_BytecodeMix/Dyn                  78.6 ns         78.6 ns      8680200
BM_Bytecode/Base                     149 ns          149 ns      4698050
BM_BytecodeMix/Base                  104 ns          104 ns      6717815
```

Bytecode Interpreter: Optimization

```
Instruction
Fibonacci[] = {
  {O_Par, 0, 0},
  {O_Int, 1, 0},
  {O_Int, 2, 1},
  {O_Jze, 0, 7},
  {O_Mov, 3, 1},
  {O_Add, 1, 2},
  {O_Mov, 2, 3},
  {O_Int, 3, 1},
  {O_Sub, 0, 3},
  {O_Jmp, -6},
  {O_Ret, 1}
};
```

```
define i32 @evalInstruction(i32* %Args) {
  %0 = load i32, i32* %Args
  store i32 %0, i32* inttoptr (i64 93929996375120 to i32*)
  ret i32 1
}

define i32 @evalInstruction.1(i32* %Args) {
  store i32 0, i32* inttoptr (i64 93929996375124 to i32*)
  ret i32 1
}

define i32 @evalInstruction.2(i32* %Args) {
  store i32 1, i32* inttoptr (i64 93929996375128 to i32*)
  ret i32 1
}

define i32 @evalInstruction.3(i32* %Args) {
  %0 = load i32, i32* inttoptr (i64 93929996375120 to i32*)
  %tobool = icmp eq i32 %0, 0
  %cond = select i1 %tobool, i32 1, i32 7
  ret i32 %cond
}
```

Bytecode Interpreter: Optimization (cont.)

```
__stage(1) int evalInstruction(struct Instruction *PC,
                               __stage(1) int *Args) __stage(1) {
  unsigned N = 0;
  while (PC->Op != O_Jmp && PC->Op != O_Jze && PC->Op != O_Ret) {
    switch (PC->Op) {
    case O_Int: Regs[PC->Operands[0]] = PC->Operands[1]; break;
    case O_Par: Regs[PC->Operands[0]] = Args[PC->Operands[1]]; break;
    case O_Add: Regs[PC->Operands[0]] += Regs[PC->Operands[1]];
    }
    N += 1;
    PC += 1;
  }
  switch (PC->Op) {
  case O_Jmp: return N + PC->Operands[0];
  case O_Jze: return N + (Regs[PC->Operands[0]] ? PC->Operands[1] : 1);
  case O_Ret: return N;
  }
}
```

Bytecode Interpreter: Optimization (cont.)

```
Instruction
Fibonacci[] = {
  {O_Par, 0, 0},
  {O_Int, 1, 0},
  {O_Int, 2, 1},
  {O_Jze, 0, 7},
  {O_Mov, 3, 1},
  {O_Add, 1, 2},
  {O_Mov, 2, 3},
  {O_Int, 3, 1},
  {O_Sub, 0, 3},
  {O_Jmp, -6},
  {O_Ret, 1}
};
```

```
define i32 @evalInstruction(i32* %Args) {
  %0 = load i32, i32* %Args
  store i32 %0, i32* inttoptr (i64 93929996375152 to i32*)
  store i32 0, i32* inttoptr (i64 93929996375156 to i32*)
  store i32 1, i32* inttoptr (i64 93929996375160 to i32*)
  %1 = load i32, i32* inttoptr (i64 93929996375152 to i32*)
  %tobool = icmp eq i32 %1, 0
  %add76 = select i1 %tobool, i32 4, i32 10
  ret i32 %add76
}
```

Bytecode Interpreter: Optimization (cont.)

Benchmark	Time	CPU	Iterations
BM_Bytecode/Dyn	87.0 ns	87.0 ns	7976372
BM_BytecodeMix/Dyn	78.6 ns	78.6 ns	8680200
BM_Bytecode/Base	149 ns	149 ns	4698050
BM_BytecodeMix/Base	104 ns	104 ns	6717815
BM_Bytecode/Opt	140 ns	140 ns	5031333
BM_BytecodeMix/Opt	36.4 ns	36.4 ns	19318499

Summary

Limitations

1. Residual code cannot be executed stand-alone (addresses of static parameters, global variables, and internal functions).
2. Binding-time annotations may be quite excessive.
3. Binding-time diagnostics are reported in terms of IR code.
4. Works best in cases with no dynamic control in the interpreter. Otherwise may require a different driver function.

Future Work

1. Apply to a "real" program.
2. Implement proof-of-concept front-end for another language.
3. Add missing binding-time rules and prove correctness of the transformation.
4. Infer annotations inter-procedurally.

Questions

https://github.com/eush77/llvm.mix
https://github.com/eush77/clang.mix
https://github.com/eush77/mix-examples

Bonus Slides

Entry Point Function

- Each intermediate execution stage builds a new `Module` in the provided `LLVMContext`.
- *Entry-point* function:
  - Creates `Module`, `IRBuilder`
  - Creates `IRBuilder`
  - Creates declarations of all used external functions and variables
  - Creates types, constants, and metadata kinds in the provided `LLVMContext`
- Entry-point function stores all the references in the *context table* that is loaded from by functions that are building the IR

Static Return Values

- If function F returns a value of stage $N-1$, $\text{StageFunction}_N(F)$ returns that value in a struct so that callers can use that value in stage $N-1$.

```
declare i32 @F(i32 %x, stage(1) %y) stage(1)

; G = StageFunction_1(F)
declare { i32, %struct.LLVMOpaqueValue* } @G(i8** %mix.context, i32 %x)
```

- Returning a $stage < N - 1$ value is not supported since static return value and the residual function are returned in the same stage.

Basic Blocks and Phi Nodes

$$\frac{B' \in succ(B)}{stage(term(B)) = stage(B')} \text{ (BasicBlock)}$$

- Basic blocks have the same stages as terminators in predecessor blocks:
  - A terminator can't jump to a block that doesn't exist yet
  - A basic block that is not a target of any jump is unreachable and can be removed

$$\frac{\varphi \in \Phi(B)}{stage(\varphi) = stage(B)} \text{ (Phi)}$$

- Phi nodes have the same stages as their basic blocks:
  - All phi nodes must be resolved at the start of a block
  - Phi nodes are meaningless without their basic block

Operand Congruence

$$\frac{op \neq \varphi}{opstage(op(\dots)) = stage(op(\dots))} \qquad \frac{a = \varphi(\dots) \qquad opstage(a) = N}{opstage(\varphi(\dots, a, \dots)) \geq N}$$

$$\frac{op \neq \varphi}{stage(op(\dots, a, \dots)) \geq opstage(a)} \text{ (Operand)}$$

- Binding time of an instruction is constrained by binding times of its operands.
- Operands of a stage-N instruction must be computed at stage N or before:

```
%1 = add i32 %0, 1   ; stage(1)  ⇒  %1 = call %struct.Value* @LLVMBuildAdd
%2 = mul i32 %1, 3   ; stage(0)      %2 = mul i32 ???, 3
```

- If phi node `%p` is an operand of `%a`, `stage(%a)` is constrained by the stage of phi's incoming value (`opstage(%p)`), not that of its basic block.

Phi Nodes and Operand Congruence

$$\frac{op \neq \varphi}{stage(op(\ldots, a, \ldots)) \geq opstage(a)} \text{(OPERAND)}$$

Operand congruence BTA rule does not apply to phi nodes: stages of phi nodes and their incoming values are unrelated:

```
A:                              ; stage(1)
  ; opstage(x) = 0 < 1 = stage(x)
  %x = phi i32 [ 0, %0 ],
               [ 1, %1 ]
```

Example of source IR

```
A:                                                   ; stage(0)
  %x = call %struct.LLVMOpaqueValue* @LLVMBuildLoad
  br label %C
B:                                                   ; stage(0)
  %y = call %struct.LLVMOpaqueValue* @LLVMBuildLoad
  br label %C
C:                                                   ; stage(0)
  ; stage(z) = 0 < 1 = opstage(z)
  %z = phi i32 [ %x, %A ], [ %y, %B ]
```

Example of stage(0) IR

Field Annotations

- Stages of struct fields are declared by field annotations.
- Calls to intrinsic `@llvm.object.stage` are generated on every access to annotated fields.

```
struct S {
  int X __stage(1);
  struct S *L
        __stage(0);
};

void
f(struct S *N)
__stage(1) {
  /* ... */
  N = N->L;
  /* ... */
  N->X = 1;
  /* ... */
}
```

```
define void @f(%struct.S* %N) {
  ; ...
  %L = getelementptr inbounds %struct.S, %struct.S* %N,
                              i32 0, i32 1
  %L1 = call %struct.S** @llvm.object.stage(%struct.S** %L,
                                             i32 0)
  %1 = load %struct.S*, %struct.S** %L1
  ; ...
  %X = getelementptr inbounds %struct.S, %struct.S* %1,
                              i32 0, i32 0
  %X2 = call i32* @llvm.object.stage(i32* %X, i32 1)
  store i32 1, i32* %X2
  ; ...
}
```

Memory Access

$$\frac{\mathit{objectstage}(p) = N}{\mathit{stage}(\mathit{load}(p)) \geq N} \text{ (Load)} \qquad \frac{\mathit{objectstage}(p) = N}{\mathit{stage}(\mathit{store}(x, p)) = N - 1} \text{ (Store)}$$

- An object annotated with object stage N is constant from this stage on, hence:
  - It is safe to `load` such an object at stage N
  - The latest stage for a `store` to the object is $N - 1$
- It is easy to ruin the correctness property of specialized code with incorrect annotations.

Wrappers for Base Types

```
struct StaticChar {
  char Val;
} __attribute__((packed,staged));

struct StaticDouble {
  double Val;
} __attribute__((packed,staged));

// ...
(__stage(1) struct StaticChar *SStr) {
  SStr->Val;  // -> object_stage=1
}
```

External vs Inline Annotations

```
%List = type {
   static i32,         ;length
   static %ListCell*,   ;head
   static %ListCell*    ;tail
}

%ListCell = type {
   dynamic i8*,         ;data
   static %ListCell*    ;next
}
```

```
struct List {
  __stage(0) unsigned length;
  __stage(0) ListCell  *head;
  __stage(0) ListCell  *tail;
};

struct ListCell {
  __stage(1) void      *data;
  __stage(0) ListCell  *next;
};
```

Recursive Descent

```
__stage(1) const char *
parseAlternative(Alternative *A, __stage(1) const char *Str) __stage(1) {
  for (Symbol *S = A->Sym; S != A->Sym + A->NSym; ++S)
    if (!(Str = parseSymbol(S, Str)))                         // dynamic control
      return NULL;
  return Str;
}
__stage(1) const char *
parseNonterminal(Nonterminal *N, __stage(1) const char *Str) __stage(1) {
  for (Alternative *A = N->Alt; A != N->Alt + N->NAlt; ++A)
    if ((const char *End = parseAlternative(A, Str))) // dynamic control
      return End;
  return NULL;
}
__stage(1) const char *
parseSymbol(Symbol *S, __stage(1) const char *Str) __stage(1) {
  switch (S->T) {
  case T_Terminal: return parseTerminal(S->Node, Str);
  case T_Nonterminal: return parseNonterminal(S->Node, Str);
  }
}
```

Recursive Descent: Avoiding Dynamic Control

```
__stage(1) const char *
parseAlternative(Alternative *A, __stage(1) const char *Str) __stage(1) {
  const char *(*Sym[A->NSym])(const char *);
  for (unsigned SNum = 0; SNum < A->NSym; ++SNum)
    Sym[SNum] = __builtin_mix_call(parse, A->Sym + SNum);
  for (unsigned SNum = 0; SNum < A->NSym; ++SNum) {
    if (!(Str = Sym[SNum](Str)))
      return NULL;
  }
  return Str;
}
__stage(1) const char *
parseNonterminal(Nonterminal *N, __stage(1) const char *Str) __stage(1) {
  const char *(*Alt[N->NAlt])(const char *);
  for (unsigned ANum = 0; ANum < N->NAlt; ++ANum)
    Alt[ANum] = __builtin_mix_call(parseAlternative, N->Alt + ANum);
  for (unsigned ANum = 0; ANum < N->NAlt; ++ANum)
    if ((const char *End = Alt[ANum](Str)))
      return End;
  return NULL;
}
```

Recursive Descent: Avoiding Dynamic Control

Did you notice infinite recursion?

```
__stage(1) const char *
parseAlternative(Alternative *A, __stage(1) const char *Str) __stage(1) {
  const char *(*Sym[A->NSym])(const char *);
  for (unsigned SNum = 0; SNum < A->NSym; ++SNum)
    Sym[SNum] = __builtin_mix_call (parse, A->Sym + SNum);
  for (unsigned SNum = 0; SNum < A->NSym; ++SNum) {
    if (!(Str = Sym[SNum](Str)))
      return NULL;
  }
  return Str;
}
__stage(1) const char *
parseNonterminal(Nonterminal *N, __stage(1) const char *Str) __stage(1) {
  const char *(*Alt[N->NAlt])(const char *);
  for (unsigned ANum = 0; ANum < N->NAlt; ++ANum)
    Alt[ANum] = __builtin_mix_call (parseAlternative, N->Alt + ANum);
  for (unsigned ANum = 0; ANum < N->NAlt; ++ANum)
    if ((const char *End = Alt[ANum](Str)))
      return End;
  return NULL;
}
```

Recursive Descent: Compiling Ahead-of-Time

```
__stage(1) void *compileNonterminal(Nonterminal *N) __stage(1) {
  return N->NAlt==1 ? N->Alt->Parse : __builtin_mix_call(parseNonterminal, N);
}
__stage(1) void *compileSymbol(Symbol *S) __stage(1) {
  switch (S->T) {
  case T_Terminal: return __builtin_mix_call(parseTerminal, S->Node);
  case T_Nonterminal: return compileNonterminal(S->Node);
  }
}
__stage(1) const char *
parse(unsigned NSym, Symbol Syms[NSym], unsigned NAlt, Alternative Alts[NAlt],
      Symbol *Start, __stage(1) const char *Str) __stage(1) {
  for (Alternative *A = Alts; A != Alts + NAlt; ++A)
    A->Parse = A->NSym == 1 ?
        NULL : __builtin_mix_call(parseAlternative, A);
  for (Symbol *S = Syms; S != Syms + NSym; ++S)
    S->Parse = compileSymbol(S);
  for (Alternative *A = Alts; A != Alts + NAlt; ++A)
    if (A->NSym == 1)
      A->Parse = A->Sym->Parse;
  return Start->Parse(Str);
}
```

Recursive Descent: Mixable Implementation

```
__stage(1) const char *
parseAlternative(Alternative *A, __stage(1) const char *Str) __stage(1) {
  for (Symbol *S = A->Sym; S != A->Sym + A->NSym; ++S) {
    if (!(Str = S->Parse(Str)))
      return NULL;
  }
  return Str;
}
__stage(1) const char *
parseNonterminal(Nonterminal *N, __stage(1) const char *Str) __stage(1) {
  for (Alternative *A = N->Alt; A != N->Alt + N->NAlt; ++A) {
    if ((const char *End = A->Parse(Str)))
      return End;
  }
  return NULL;
}
```

Recursive Descent: Mixable Implementation (cont.)

```
----------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
----------------------------------------------------------------------
BM_RecursiveDescentInt/Base          87.5 ns         87.5 ns      8005007
BM_RecursiveDescentMix/Base          52.2 ns         52.2 ns     13398241
BM_RecursiveDescentInt/Unroll        65.1 ns         65.1 ns     10722491
BM_RecursiveDescentMix/Unroll        34.8 ns         34.8 ns     20187461
```