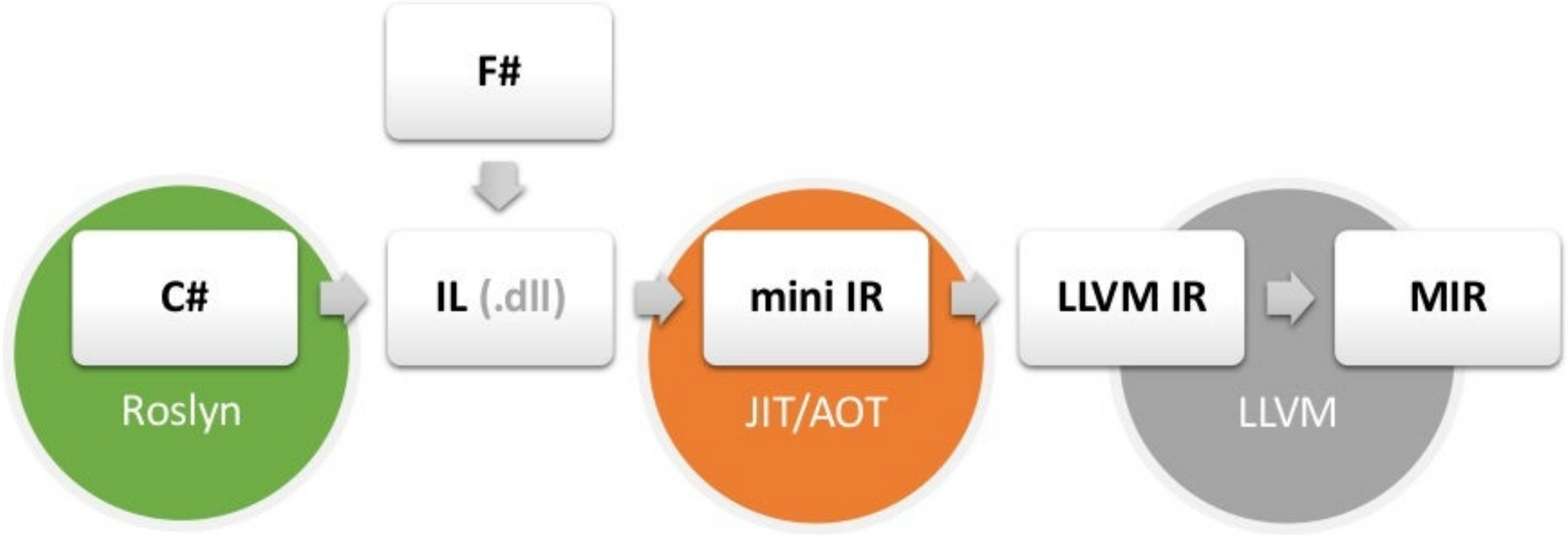
LLVM back-end for C#

@EgorBo
Senior Software Engineer at Microsoft

Mono-LLVM: quick intro



LLVM: C# to MC


```
int MyAdd(int a, int b)
{
    return a + b;
}
```



```
.method private hidebysig static
        int32 MyAdd (
            int32 a,
            int32 b
        ) cil managed
    {
        IL_0000: ldarg.0
        IL_0001: ldarg.1
        IL_0002: add
        IL_0003: ret
    }
```

LLVM: C# to MC


```
.method private hidebysig static
       int32 MyAdd (
           int32 a,
           int32 b
       ) cil managed
   {
       IL_0000: ldarg.0
       IL_0001: ldarg.1
       IL_0002: add
       IL_0003: ret
   }
```



```
define monocc i32 @"Program:MyAdd (int,int)"
   (i32 %arg_a, i32 %arg_b) {
BB0:
    br label %BB3

BB3:
    br label %BB2

BB2:
    %t22 = add i32 %arg_a, %arg_b
    br label %BB1

BB1:
    ret i32 %t22
}
```

LLVM: C# to MC


```
.method private hidebysig static
       int32 MyAdd (
           int32 a,
           int32 b
       ) cil managed
   {
       IL_0000: ldarg.0
       IL_0001: ldarg.1
       IL_0002: add
       IL_0003: ret
   }
```



```
define monocc i32 @"Program:MyAdd (int,int)"
   (i32 %arg_a, i32 %arg_b) {
BB0:
    br label %BB3

BB3:                                    (simplifycfg)
    br label %BB2

BB2:
    %t22 = add i32 %arg_a, %arg_b
    br label %BB1

BB1:
    ret i32 %t22
}
```

LLVM: C# to MC


```
define monocc i32 @"Program:MyAdd (int,int)"(i32 %arg_a, i32 %arg_b) #0
{
BB0:
    %t22 = add i32 %arg_a, %arg_b
    ret i32 %t22
}
```


(I am skipping the machine IR part)

```
lea eax, [rdi + rsi]
ret
```

LLVM: C# to MC

```
void MySet(byte* array, int len, byte val)
{
    for (int i = 0; i < len; i++)
        array[i] = val;
}
```



```
define monocc void @"MySet"(i8* %arg_array,
                            i32 %arg_len, i32 %arg_val) #0 {
BB0:
  br label %BB3

BB3:
  br label %BB2

BB2:
  br label %BB4

BB4:
  %0 = phi i32 [ 0, %BB2 ], [ %t32, %BB5 ]
  %1 = icmp slt i32 %0, %arg_len
  br i1 %1, label %BB5, label %BB6

BB6:
  br label %BB1

BB5:
  %t23 = sext i32 %0 to i64
  %3 = getelementptr i8, i8* %arg_array, i64 %t23
  %4 = trunc i32 %arg_val to i8
  store i8 %4, i8* %3
  %t32 = add i32 %0, 1
  br label %BB4

BB1:
  ret void
}
```

```
define monocc void @"MySet"(i8* %arg_array,
             i32 %arg_len, i32 %arg_val) #0 {
BB0:
  br label %BB3

BB3:
  br label %BB2

BB2:
  br label %BB4

BB4:
  %0 = phi i32 [ 0, %BB2 ], [ %t32, %BB5 ]
  %1 = icmp slt i32 %0, %arg_len
  br i1 %1, label %BB5, label %BB6

BB6:
  br label %BB1

BB5:
  %t23 = sext i32 %0 to i64
  %3 = getelementptr i8, i8* %arg_array, i64 %t23
  %4 = trunc i32 %arg_val to i8
  store i8 %4, i8* %3
  %t32 = add i32 %0, 1
  br label %BB4

BB1:
  ret void
}
```

opt

```
define monocc void @"MySet"(i8* %arg_array,
             i32 %arg_len, i32 %arg_val) #0 {
BB0:
  %0 = icmp sgt i32 %arg_len, 0
  br i1 %0, label %BB5.lr.ph, label %BB1

BB5.lr.ph:
  %1 = trunc i32 %arg_val to i8
  %2 = zext i32 %arg_len to i64
  call void @llvm.memset.p0i8.i64(i8* %arg_array, i8 %1, i64 %2…)
  br label %BB1

BB1:
  ret void
}
```

LLVM needs hints from front-ends

- Alias-analysis (`noalias`, TBAA, etc)
- PGO-data and @llvm.expect (branch-weights)
- Use **GEP**s where possible instead of `ptrtoint+add+inttoptr`
- Language specific order of optimization passes (maybe even custom passes)
- Alignment hints
- FaultMaps and implicit null-checks
- Use LLVM intrinsics where needed (libcalls, HW intrinsics, etc)
- Don’t forget about Fast-Math!
- NIT: Nullability, Escape analysis, etc

AA is important!

A ▾ Save/Load + Add new... ▾ Vim C ▾

```
1
2
3   void MyCopy(int* restrict a, int* b, int len)
4   {
5       for (int i=0; i<len; i++)
6       {
7           a[i] = b[i];
8       }
9   }
```

x86-64 clang 9.0.0 ▾ -g0 -emit-llvm -O2

A ▾ 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle Libraries ▾

```
1   target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
2   target triple = "x86_64-unknown-linux-gnu"
3
4   define dso_local void @MyCopy(i32* noalias nocapture, i32* nocapture readonly, i32) local_unnamed_a
5     %4 = icmp sgt i32 %2, 0
6     br i1 %4, label %5, label %10
7
8   5: ; preds = %3
9     %6 = bitcast i32* %1 to i8*
10    %7 = bitcast i32* %0 to i8*
11    %8 = zext i32 %2 to i64
12    %9 = shl nuw nsw i64 %8, 2
13    call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 4 %7, i8* align 4 %6, i64 %9, i1 false)
14    br label %10
15
16  10: ; preds = %5, %3
17    ret void
18  }
19
```

LLVM: ffast-math

1
```
float result = (x / 10) + 1 + 1;
```
2
```
float result = (x / 10) + 2;
```
3
```
float result = (x * 0.1) + 2;
```
4
```
float result = fmadd(x, 0.1, 2);
```

CoreCLR, Mono, C++ -O2

```
vdivsd xmm0, xmm0, QWORD PTR .LC0[rip]
vaddsd xmm0, xmm0, xmm1
vaddsd xmm0, xmm0, xmm1
```

Mono-LLVM (--llvm --ffast-math)

```
vfmadd132sd xmm0, xmm1, QWORD PTR .LC0[rip]
```

LLVM: ffast-math

- No NaNs (NOTE: some apps rely on NaN values, e.g. WPF)
- No Infs
- No Signed Zeros (e.g. **Math**.Max(-0.0, 0.0) will return -0.0)
- Approximate functions
- Recognize FMA patterns (a * b + c)
- Reassociation transformations (x + c1 + c2 => x + (c1 + c2))

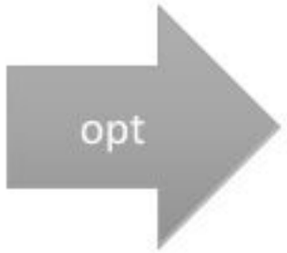
C# Math/MathF are LLVM intrinsics!


```
static float Foo(float x)
{
    return MathF.Sqrt(x) * MathF.Sqrt(x);
}
```


```
define float @"Foo“ (float %arg_x) {
BB0:
  br label %BB3

BB3:
  br label %BB2

BB2:
  %t19 = call fast float @llvm.sqrt.f32(float %arg_x)
  %t21 = call fast float @llvm.sqrt.f32(float %arg_x)
  %t22 = fmul fast float %t19, %t21
  br label %BB1

BB1:
  ret float %t22
}

declare float @llvm.sqrt.f32(float)
```

C# Math/MathF are LLVM intrinsics!


```
define float @"Foo“ (float %arg_x) {
BB0:
  br label %BB3

BB3:
  br label %BB2

BB2:
  %t19 = call fast float @llvm.sqrt.f32(float %arg_x)
  %t21 = call fast float @llvm.sqrt.f32(float %arg_x)
  %t22 = fmul fast float %t19, %t21
  br label %BB1

BB1:
  ret float %t22
}

declare float @llvm.sqrt.f32(float)
```




```
define float @"Foo“ (float %arg_x) {
BB0:
  ret float %arg_x
}
```

C# Math/MathF are LLVM intrinsics!


```
Math.Abs(X)  * Math.Abs(X)   => X * X
Math.Sin(X)  / Math.Cos(X)   => Math.Tan(X)
Math.Sqrt(X) * Math.Sqrt(X)  => X
Math.Sin(-X)                 => -Math.Sin(X)

MathF.Pow(X, 0.5f)           => MathF.Sqrt(X)
MathF.Pow(X, 2)              => X * X
MathF.Pow(X, 4)              => (X * X) * (X * X)
```

Significant boost in some FP benchmarks:

Benchmark	Fast-Math	No Fast-Math
Benchstone.BenchF.BenchMk2.Test	1.00	2.25
Benchstone.BenchF.BenchMrk.Test	1.00	2.37
Benchstone.BenchF.Lorenz.Test	1.00	1.85
Burgers.Test0	1.00	1.39
Burgers.Test1	1.00	2.30
Burgers.Test2	1.00	2.32
System.MathBenchmarks.Double.Cosh	1.00	1.50

(CPU without FMA)

One of those benchmarks (Lorenz equations):


```
            x_arg = s_x + hdiv2 * k2;
            y_arg = s_y + hdiv2 * l2;
            z_arg = s_z + hdiv2 * m2;

            k3 = F(t_arg, x_arg, y_arg, z_arg);
            l3 = G(t_arg, x_arg, y_arg, z_arg);
            m3 = H(t_arg, x_arg, y_arg, z_arg);

            t_arg = s_t + s_h;
            x_arg = s_x + s_h * k3;
            y_arg = s_y + s_h * l3;
            z_arg = s_z + s_h * m3;

            k4 = F(t_arg, x_arg, y_arg, z_arg);
            l4 = G(t_arg, x_arg, y_arg, z_arg);
            m4 = H(t_arg, x_arg, y_arg, z_arg);

            s_x = s_x + hdiv6 * (k1 + 2.0 * k2 + 2.0 * k3 + k4);
            s_y = s_y + hdiv6 * (l1 + 2.0 * l2 + 2.0 * l3 + l4);
            s_z = s_z + hdiv6 * (m1 + 2.0 * m2 + 2.0 * m3 + m4);
            s_t = t_arg;
        }

        return true;
    }

    private static double F(double t, double x, double y, double z)
    {
        return (10.0 * (y - x));
    }

    private static double G(double t, double x, double y, double z)
    {
        return (x * (28.0 - z) - y);
    }

    private static double H(double t, double x, double y, double z)
    {
        return (x * y - (8.0 * z) / 3.0);
    }
```

Don’t forget to feed the **LLVM!**
with some **PGO** data for better codegen

```
static int Max(int x, int y)
{
    return x > y ? x : y;
}
```

Default

```
define i32 @Max(i32 %0, i32 %1) {
  %3 = icmp sgt i32 %0, %1
  %4 = select i1 %3, i32 %0, i32 %1
  ret i32 %4
}
```

```
mov eax, esi
cmp edi, esi
cmovge eax, edi
ret
```

With profile data (PGO)

```
define i32 @Max(i32 %0, i32 %1) {
  %3 = icmp sgt i32 %0, %1 !prof 0
  %4 = select i1 %3, i32 %0, i32 %1
  ret i32 %4
}

!0 = !{!"branch_weights", i32 1000, i32 1}
```

```
mov eax, edi
cmp edi, esi
jle .LBB1_1
ret
.LBB1_1:
mov eax, esi
ret
```

COMPILER EXPLORER

Add... More

LLVM IR source #1

Save/Load Add new... Vim LLVM IR

```
define i32 @Max(i32, i32) {
  %3 = icmp sgt i32 %0, %1
  %4 = select i1 %3, i32 %0, i32 %1
  ret i32 %4
}
```

opt (trunk) (Editor #1, Compiler #2) LLVM IR

opt (trunk) -pgo-instr-gen -instrprof

11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle Libraries Add new... Add tool...

```
define i32 @Max(i32 %0, i32 %1) {
  %pgocount = load i64, i64* getelementptr inbounds ([2 x i64], [2 x i64]* @__profc_Max, i64 0, i64 0)
  %3 = add i64 %pgocount, 1
  store i64 %3, i64* getelementptr inbounds ([2 x i64], [2 x i64]* @__profc_Max, i64 0, i64 0)
  %4 = icmp sgt i32 %0, %1
  %5 = zext i1 %4 to i64
  %pgocount1 = load i64, i64* getelementptr inbounds ([2 x i64], [2 x i64]* @__profc_Max, i64 0, i64 1)
  %6 = add i64 %pgocount1, %5
  store i64 %6, i64* getelementptr inbounds ([2 x i64], [2 x i64]* @__profc_Max, i64 0, i64 1)
  %7 = select i1 %4, i32 %0, i32 %1
  ret i32 %7
}

declare void @llvm.instrprof.increment(i8*, i64, i32, i32) #0

declare void @llvm.instrprof.increment.step(i8*, i64, i32, i32, i64) #0

define linkonce_odr hidden i32 @__llvm_profile_runtime_user() #1 comdat {
  %1 = load i32, i32* @__llvm_profile_runtime
  ret i32 %1
}

define internal void @__llvm_profile_register_functions() unnamed_addr {
  call void @__llvm_profile_register_function(i8* bitcast ({ i64, i64, i64*, i8*, i8*, i32, [2 x i16] }* @__profd_Max to i8*))
  call void @__llvm_profile_register_names_function(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @__llvm_prf_nm, i32 0, i32 0), i64
  ret void
}

declare void @__llvm_profile_register_function(i8*)

declare void @__llvm_profile_register_names_function(i8*, i64)

define internal void @__llvm_profile_init() unnamed_addr #1 {
  call void @__llvm_profile_register_functions()
  ret void
}

attributes #0 = { nounwind }
attributes #1 = { noinline }
```

PGO and switch


```
char* format(char format)
{
    switch (format)
    {
        case 'G':
            return defaultFormat();

        case 'X':
            return hexFormat();

        case 'f':
            return floatFormat();

        case 'p':
            return percentFormat();
    }
    return defaultFormat();
}
```

```
format(int): # @format(int)
  cmp edi, 101
  jg .LBB0_4
  cmp edi, 71
  je .LBB0_8
  cmp edi, 88
  jne .LBB0_8
  jmp hexFormat() # TAILCALL
.LBB0_4:
  cmp edi, 102
  je .LBB0_9
  cmp edi, 112
  jne .LBB0_8
  jmp percentFormat() # TAILCALL
.LBB0_8:
  jmp defaultFormat() # TAILCALL
.LBB0_9:
  jmp floatFormat() # TAILCALL
```

PGO and switch


```
char* format(char format)
{
    switch (__builtin_expect(format, 'X'))
    {
        case 'G’:
            return defaultFormat();

        case 'X’:
            return hexFormat();

        case 'f’:
            return floatFormat();

        case 'p’:
            return percentFormat();
    }
    return defaultFormat();
}
```

```
format(int): # @format(int)
  movsxd rax, edi
  cmp rax, 88
  jne .LBB0_2
  jmp hexFormat() # TAILCALL
.LBB0_2:
  cmp rax, 112
  je .LBB0_6
  cmp rax, 102
  je .LBB0_7
  cmp rax, 71
  jmp defaultFormat() # TAILCALL
.LBB0_6:
  jmp percentFormat() # TAILCALL
.LBB0_7:
  jmp floatFormat() # TAILCALL
```

PGO and Guarded Devirtualization


```
static void Foo(IAnimal animal)
{
    animal.MakeSound();
}
```

```
static void Foo(IAnimal animal)
{
    if (animal is Dog dog)
        dog.Bark();
    else
        animal.MakeSound();
}
```

Mono-LLVM

- We use github.com/dotnet/llvm-project fork
- Currently target LLVM 6 but on our way to LLVM 9
- We currently use LLVM for:
  - AOT (opt + llc)
  - JIT (legacy::PassManager, ORCv1)

Mono-LLVM

- `opt –O2` is only for C++
- We have a lot of additional checks: null-checks, bound-checks
- We have to insert safe-points (-place-safepoints) for non-preemptive mode

```
void MyMethod(byte* array, int len, byte val)
{
    if (unlikely(gcRequested))
        performGC();

    for (int i = 0; i < len; i++)
    {
        if (unlikely(gcRequested))
            performGC();

        array[i] = val;
    }

    if (unlikely(gcRequested))
        performGC();
}
```

How do we use optimizations?

- **LLVM AOT:**
  opt -O2 –place-safepoints

`**opt –O2**`: These pass pipelines make a good starting point for an optimizing compiler for any language, but they have been carefully tuned for C and C++, not your target language.

- **LLVM JIT:**
  PassManager with a small subset of the most useful passes in a specific order:

```
-simplifycfg
-sroa
-lower-expect
-instcombine
-licm
-simplifycfg
-lcssa
-indvars
-loop-deletion
-gvn
-memcpyopt
-sccp
-bdce
-instcombine
-dse
-simplifycfg
```

Hide 'no-op' passes | Open current diff in kdiff3 | Filter by PassName | Filter by Func

- gc
- mono_aot_HelloWorld_icall_cold_wrapper_265
- llvm_code_start
- mono_aot_HelloWorldinit_method
- mono_aot_HelloWorldinit_method_gshared_mrgctx
- mono_aot_HelloWorldinit_method_gshared_this
- mono_aot_HelloWorldinit_method_gshared_vtable
- PP_Main
- TimeSpanTokenizer__ctor_System_ReadOnlySpan_1_char
- TimeSpanTokenizer__ctor_System_ReadOnlySpan_1_char_int
- TimeSpanTokenizer_get_NextChar
  - Module Verifier -verify
  - Instrument function entry/exit with calls to e.g. mcount() (pre inlining) -??
  - **Simplify the CFG** -simplifycfg
  - SROA -sroa
  - **Early CSE** -early-cse
  - Lower 'expect' Intrinsics -lower-expect
  - Promote Memory to Register -mem2reg
  - **Combine redundant instructions** -instcombine
  - **Simplify the CFG** -simplifycfg
  - Remove unused exception handling info -prune-eh
  - Function Integration/Inlining -inline
  - Deduce function attributes -functionattrs
  - Remove unused exception handling info -prune-eh
  - Function Integration/Inlining -inline
  - Deduce function attributes -functionattrs
  - SROA -sroa
  - Early CSE w/ MemorySSA -memoryssa
  - Speculatively execute instructions if target has divergent branches -??
  - Jump Threading -jump-threading
  - Value Propagation -correlated-propagation
  - Simplify the CFG -simplifycfg
  - **Combine redundant instructions** -instcombine
  - Conditionally eliminate dead library calls -??
  - PGOMemOPSize -??
  - Tail Call Elimination -tailcallelim
  - Simplify the CFG -simplifycfg
  - Reassociate expressions
  - Canonicalize natural loops -loop-simplify
  - LCSSA Verifier -??
  - Loop-Closed SSA Form Pass -lcssa
  - Simplify the CFG -simplifycfg
  - **Combine redundant instructions** -instcombine
  - Canonicalize natural loops -loop-simplify
  - LCSSA Verifier -??
  - Loop-Closed SSA Form Pass -lcssa
  - MergedLoadStoreMotion -??
  - Global Value Numbering -gvn
  - MemCpy Optimization -memcpyopt
  - Sparse Conditional Constant Propagation -sccp
  - Demanded bits analysis -??
  - Bit-Tracking Dead Code Elimination -bdce
  - Combine redundant instructions -instcombine
  - Jump Threading -jump-threading
  - Value Propagation -correlated-propagation
  - Dead Store Elimination -dse

```
; Function Attrs: noinline uwtable
define hidden monocc i16 @TimeSpanTokenizer_get_NextChar(i64* %this) #4 {
BB0:
  %0 = icmp eq i64* %this, null
  br i1 %0, label %EX_BB3, label %NOEX_BB4, !make.implicit !2
EX_BB3:                                           ; preds = %BB0
  call void @p_1_plt__jit_icall_llvm_throw_corlib_exception_abs_trampoline_llvm(i32 204)
  unreachable
NOEX_BB4:                                         ; preds = %BB0
  %1 = getelementptr i64, i64* %this, i64 2
  %2 = bitcast i64* %1 to i32*
  %t22 = load i32, i32* %2, align 4
  %t19 = add i32 %t22, 1
  store i32 %t19, i32* %2, align 4
  %3 = getelementptr i64, i64* %this, i64 1
  %4 = bitcast i64* %3 to i32*
  %t30 = load i32, i32* %4, align 4, !range !17
  %5 = icmp ult i32 %t19, %t30
  br i1 %5, label %BB4, label %BB1
BB4:                                              ; preds = %NOEX_BB4
  %t42 = sext i32 %t19 to i64
  %6 = zext i32 %t30 to i64
  %7 = icmp ugt i64 %6, %t42
  br i1 %7, label %NOEX_BB13, label %EX_BB12
EX_BB12:                                          ; preds = %BB4
  call void @p_1_plt__jit_icall_llvm_throw_corlib_exception_abs_trampoline_llvm(i32 166)
  unreachable
NOEX_BB13:                                        ; preds = %BB4
  %t35 = load i64, i64* %this, align 8
  %t36 = shl nsw i64 %t42, 1
  %t37 = add i64 %t35, %t36
  %8 = inttoptr i64 %t37 to i16*
  %9 = load i16, i16* %8, align 2
  br label %BB1
BB1:                                              ; preds = %NOEX_BB4, %NOEX_BB13
  %10 = phi i16 [ %9, %NOEX_BB13 ], [ 0, %NOEX_BB4 ]
  ret i16 %10
}
```

```
; Function Attrs: noinline uwtable
define hidden monocc i16 @TimeSpanTokenizer_get_NextChar(i64* %this) #4 {
BB0:
  %0 = icmp eq i64* %this, null
  br i1 %0, label %EX_BB3, label %NOEX_BB4, !make.implicit !2
EX_BB3:                                           ; preds = %BB0
  call void @p_1_plt__jit_icall_llvm_throw_corlib_exception_abs_trampoline_llvm(i32 204)
  unreachable
NOEX_BB4:                                         ; preds = %BB0
  %1 = getelementptr i64, i64* %this, i64 2
  %2 = bitcast i64* %1 to i32*
  %t22 = load i32, i32* %2, align 4
  %t19 = add i32 %t22, 1
  store i32 %t19, i32* %2, align 4
  %3 = getelementptr i64, i64* %this, i64 1
  %4 = bitcast i64* %3 to i32*
  %t30 = load i32, i32* %4, align 4, !range !17
  %5 = icmp ult i32 %t19, %t30
  br i1 %5, label %BB4, label %BB1
BB4:                                              ; preds = %NOEX_BB4
  %t42 = sext i32 %t19 to i64
  %6 = zext i32 %t30 to i64
  %7 = icmp ugt i64 %6, %t42
  br i1 %7, label %NOEX_BB13, label %EX_BB12
EX_BB12:                                          ; preds = %BB4
  call void @p_1_plt__jit_icall_llvm_throw_corlib_exception_abs_trampoline_llvm(i32 166)
  unreachable
NOEX_BB13:                                        ; preds = %BB4
  %t35 = load i64, i64* %this, align 8
  %t36 = mul i64 %t42, 2
  %t37 = add i64 %t35, %t36
  %8 = inttoptr i64 %t37 to i16*
  %9 = load i16, i16* %8, align 2
  br label %BB1
BB1:                                              ; preds = %NOEX_BB4, %NOEX_BB13
  %10 = phi i16 [ %9, %NOEX_BB13 ], [ 0, %NOEX_BB4 ]
  ret i16 %10
}
```

LLVM: tons of optimizations

- `-adce`: Aggressive Dead Code Elimination
- `-always-inline`: Inliner for `always_inline` functions
- `-argpromotion`: Promote 'by reference' arguments to scalars
- `-bb-vectorize`: Basic-Block Vectorization
- `-block-placement`: Profile Guided Basic Block Placement
- `-break-crit-edges`: Break critical edges in CFG
- `-codegenprepare`: Optimize for code generation
- `-constmerge`: Merge Duplicate Global Constants
- `-constprop`: Simple constant propagation
- `-dce`: Dead Code Elimination
- `-deadargelim`: Dead Argument Elimination
- `-deadtypeelim`: Dead Type Elimination
- `-die`: Dead Instruction Elimination
- `-dse`: Dead Store Elimination
- `-functionattrs`: Deduce function attributes
- `-globaldce`: Dead Global Elimination
- `-globalopt`: Global Variable Optimizer
- `-gvn`: Global Value Numbering
- `-indvars`: Canonicalize Induction Variables
- `-inline`: Function Integration/Inlining
- `-instcombine`: Combine redundant instructions
- `-aggressive-instcombine`: Combine expression patterns
- `-internalize`: Internalize Global Symbols
- `-ipconstprop`: Interprocedural constant propagation
- `-ipsccp`: Interprocedural Sparse Conditional Constant Propagation
- `-jump-threading`: Jump Threading
- `-lcssa`: Loop-Closed SSA Form Pass
- `-licm`: Loop Invariant Code Motion
- `-loop-deletion`: Delete dead loops
- `-loop-extract`: Extract loops into new functions
- `-loop-extract-single`: Extract at most one loop into a new function
- `-loop-reduce`: Loop Strength Reduction
- `-loop-rotate`: Rotate Loops
- `-loop-simplify`: Canonicalize natural loops
- `-loop-unroll`: Unroll loops
- `-loop-unroll-and-jam`: Unroll and Jam loops
- `-loop-unswitch`: Unswitch loops
- `-loweratomic`: Lower atomic intrinsics to non-atomic form
- `-lowerinvoke`: Lower invokes to calls, for unwindless code generators
- `-lowerswitch`: Lower `SwitchInsts` to branches
- `-mem2reg`: Promote Memory to Register
- `-memcpyopt`: MemCpy Optimization
- `-mergefunc`: Merge Functions
- `-mergereturn`: Unify function exit nodes
- `-partial-inliner`: Partial Inliner
- `-prune-eh`: Remove unused exception handling info
- `-reassociate`: Reassociate expressions
- `-reg2mem`: Demote all values to stack slots
- `-sroa`: Scalar Replacement of Aggregates
- `-sccp`: Sparse Conditional Constant Propagation
- `-simplifycfg`: Simplify the CFG
- `-sink`: Code sinking
- `-strip`: Strip all symbols from a module
- `-strip-dead-debug-info`: Strip debug info for unused symbols
- `-strip-dead-prototypes`: Strip Unused Function Prototypes
- `-strip-debug-declare`: Strip all `llvm.dbg.declare` intrinsics
- `-strip-nondebug`: Strip all symbols, except dbg symbols, from a module
- `-tailcallelim`: Tail Call Elimination

Null-checks, Bound-checks


```
static int Test(int[] array)
{
    return array[42];
}
```

Null-checks, Bound-checks


```
static int Test(int[] array)
{
    if (array == null) ThrowNRE();
    return array[42];
}
```

Null-checks, Bound-checks


```
static int Test(int[] array)
{
    if (array == null) ThrowNRE();
    if ((uint)array.Length <= 42) ThrowOBE();
    return array[42];
}
```

Null-checks, Bound-checks


```
static int Test(int[] array)
{
    if (array == null) ThrowNRE();
    if ((uint)array.Length <= 42) ThrowOBE();
    return array[42];
}
```

```
push   rax
cmp    DWORD PTR [rdi+0x18],0x0
je     1d <Test__int___+0x1d>
mov    eax,DWORD PTR [rdi+0x20]
pop    rcx
ret
movabs rax,0x1f1f030
mov    edi,0xcc
call   QWORD PTR [rax]
movabs rax,0x1f1f040
mov    edi,0xa6
call   QWORD PTR [rax]
```

LLVM Home | Documentation » Reference »

FaultMaps and implicit checks

InductiveRangeCheckElimination Pass


```
public static void Zero1000Elements(int[] array)
{
    for (int i = 0; i < 1000; i++)
        array[i] = 0; // bound checks will be inserted here
}
```

"If you language uses range checks, consider using the IRCE pass. It is not currently part of the standard pass order."

InductiveRangeCheckElimination Pass


```
public static void Zero1000Elements(int[] array)
{
    int limit = Math.Min(array.Length, 1000);

    for (int i = 0; i < limit; i++)
        array[i] = 0; // bound checks are not needed here!

    for (int i = limit; i < 1000; i++)
        array[i] = 0; // bound checks are needed here

    // so at least we could "zero" first `limit` elements without bound checks
}
```

InductiveRangeCheckElimination Pass


```
public static void Zero1000Elements(int[] array)
{
    int limit = Math.Min(array.Length, 1000);

    for (int i = 0; i < limit - 3; i += 4)
    {
        array[i] = 0;
        array[i+1] = 0;
        array[i+2] = 0;
        array[i+3] = 0;
    }

    for (int i = limit; i < 1000; i++)
        array[i] = 0; // bound checks are needed here

    // so at least we could "zero" first `limit` elements without bound checks
}
```

Now we can even unroll the first loop!

System.Runtime.Intrinsics.*


```
static uint Foo(uint x)
{
    return Lzcnt.LeadingZeroCount(x);
}
```

```
define i32 @Foo(i32 %x)
{
    %0 = call i32 @llvm.ctlz.i32(i32 %x, i1 true)
    ret i32 %0
}
```

```
; x86 with lzcnt
lzcnt eax, edi
ret

; Arm64 (AArch64)
cls w0, w0
ret
```

System.Runtime.Intrinsics.*


```
int MySum(int[] array)
{
    fixed (int* ptr = array)
    {
        var sum = Vector128<int>.Zero;
        for (int i = 0; i < array.Length; i += 4)
        {
            sum = Sse2.Add(sum, Sse2.LoadVector128(ptr + i));
        }

        sum = Ssse3.HorizontalAdd(sum, sum);
        sum = Ssse3.HorizontalAdd(sum, sum);
        return sum.ToScalar();
    }
}
```

```
G_M55440_IG07:        ;; bbWeight=4
       movsxd   r8, ecx
       vpaddd   xmm0, xmm0, xmmword ptr [rax+4*r8]
       add      ecx, 4
       cmp      edx, ecx
       jg       SHORT G_M55440_IG07

G_M55440_IG08:
       vphaddd  xmm0, xmm0, xmm0
       vphaddd  xmm0, xmm0, xmm0
       vmovapd  xmmword ptr [rsp+20H], xmm0
       mov      eax, dword ptr [rsp+20H]
```

What's wrong with this benchmark?

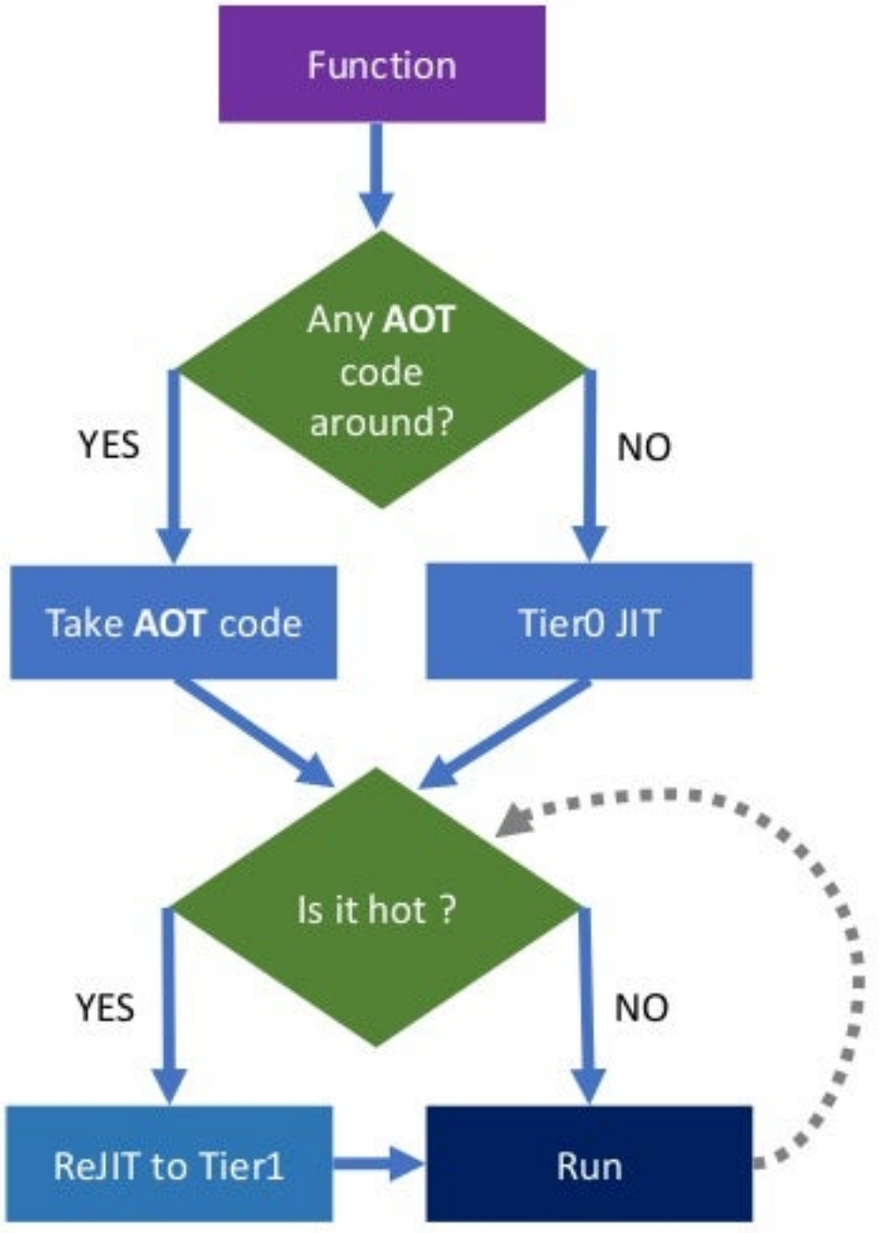

```
[Benchmark]
public bool IsNaN(float value)
{
    bool result = false;

    for (int i = 0; i < 1000000; i++)
    {
        result &= float.IsNaN(value);
        value += 1.0f;
    }

    return result;
}
```

```
[Benchmark]
public bool IsNaN(float value)
{
    return false;
}
```

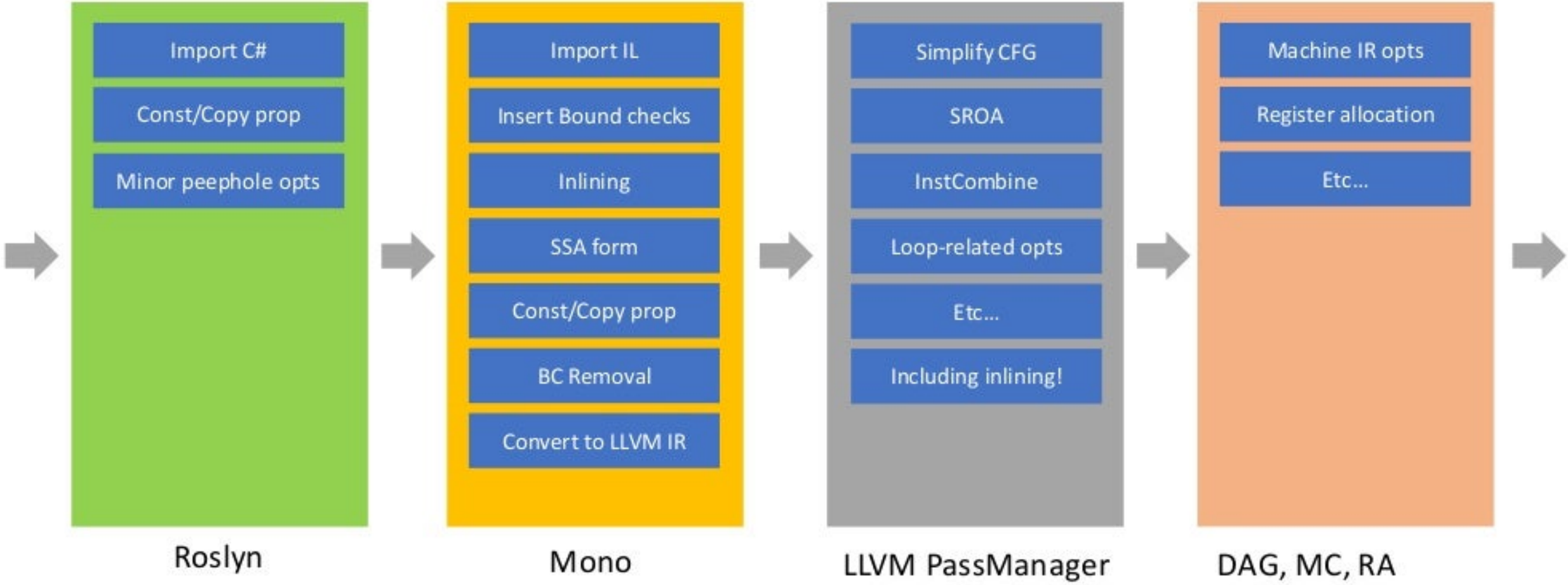
Pros & Cons

- **Pros**:
  - Tons of optimizations, frontends aren't required to emit optimized IR.
  - High MC quality and performance, especially for AArch64!
  - A lot of smart people and huge companies invest into LLVM:
    - Apple, Google, Samsung, Intel, Microsoft, etc.
- **Cons**:
  - Deadly slow to compile and optimize stuff. LLVM JIT is an oxymoron (unless it's used in a tiered compilation)
  - LLVM JIT adds 30-50mb to your application
  - Not super friendly for managed languages (Precise GC, "movable" pointers, etc)

Tiered JIT


Function
Any **AOT** code around?
YES
NO
Take **AOT** code
Tier0 JIT
Is it hot ?
YES
NO
ReJIT to Tier1
Run

Optimizations… optimizations everywhere!



Runtimes for C# (IL):

- **.NET Framework** 4.x
  - JIT
  - AOT (Fragile NGEN)
- **CoreCLR**
  - JIT (Tiered)
  - AOT (ReadyToRun – R2R)
- **Mono**
  - JIT
  - (Full)AOT
  - LLVM (AOT, FullAOT, JIT)
  - Interpreter
- **CoreRT**
- Unity **IL2CPP**
- Unity **Burst**

Q&A

twitter: EgorBo
blog: EgorBo.com

EgorBo
Senior Software Engineer at Microsoft