LLVM-backed goodies in LDC

Johan Engelen

LDC team

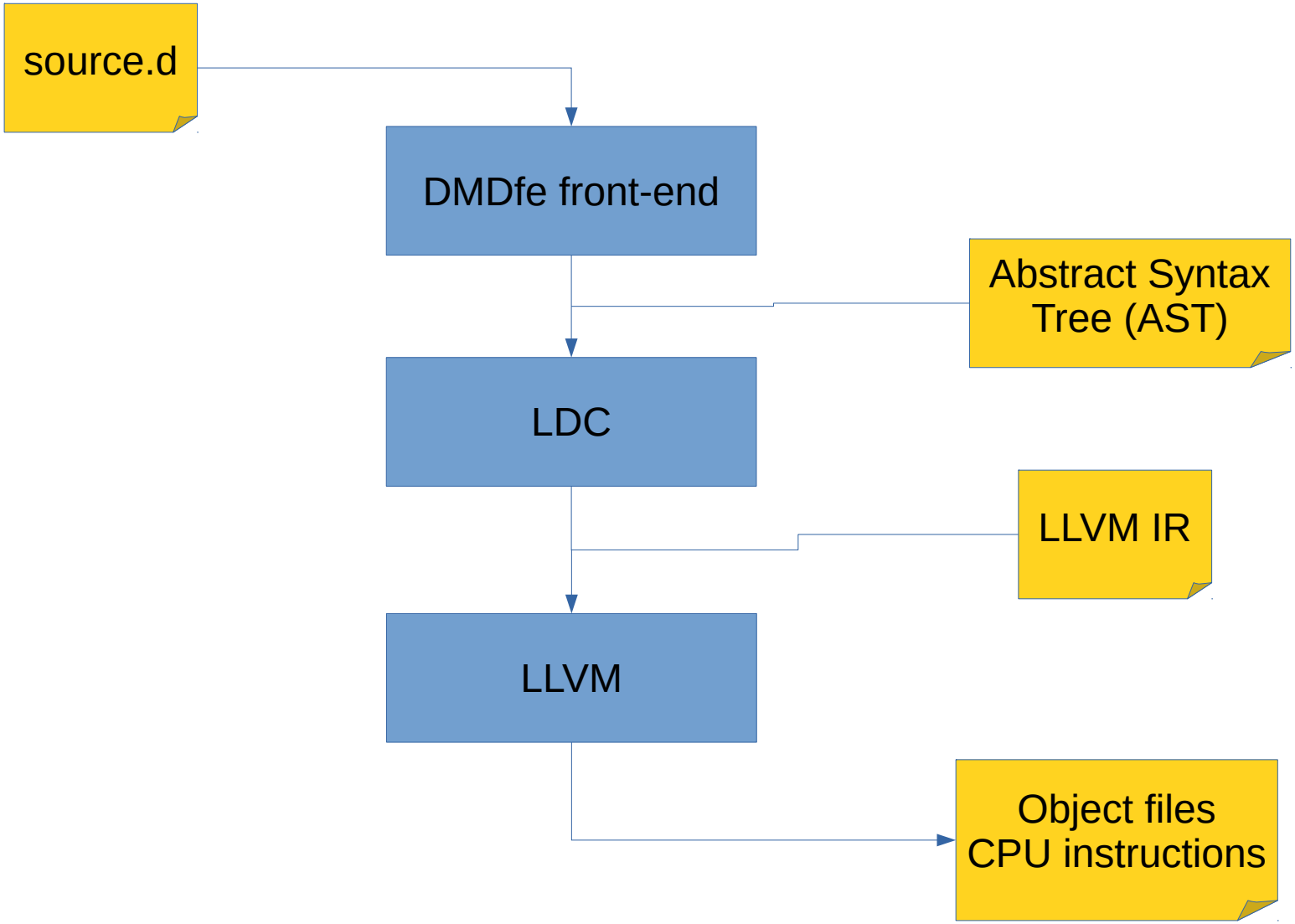
https://johanengelen.github.io

Outline

- What LLVM provides and introduction to LLVM IR
- PGO innards
- Fuzzing and ASan

Feel free to interrupt me any time for questions or comments

DMDfe – LDC – LLVM



What LLVM provides

- Machine code generation
  - Cross-target, x86, ARM, PowerPC, …, GPU and OpenCL (https://github.com/libmir/dcompute)
- Optimization
- Well-defined interface: the Intermediate Representation (IR)
- “sanitizers” and profiling
- JIT (`@dynamicCompile`)
- Linker as library (linker integrated into LDC)
- C++ compiler as library (Calypso, and recently Atila's work)
- ...

D —> LLVM IR

```d
int foo();
bool bar();

bool callBarIfFooIsEqualTo42()
{
    if (foo() == 42)
        return bar();

    return false;
}
```

```
define zeroext i1 @callBarIfFooIsEqualTo42() #1 {
  %1 = call i32 @foo() #0
  %2 = icmp eq i32 %1, 42
  br i1 %2, label %if, label %endif

if:
  %3 = call zeroext i1 @bar() #0
  ret i1 %3

dummy.afterreturn:
  br label %endif

endif:
  ret i1 false
}
```

D —> LLVM IR

```d
class A {  int field;
           int foo() { return 0; }   }
int callFoo(A a) { return a.foo() * 123; }
```

```
%dconf.A = type { [6 x i8*]*, i8*, i32 }

@_D5dconf1A6__initZ = constant %dconf.A { [6 x i8*]* @_D5dconf1A6__vtblZ,
                                          i8* null,
                                          i32 0
                                        }, align 8

; Function Attrs: norecurse nounwind readnone uwtable
define i32 @_D5dconf1A3fooMFZi(%dconf.A* nocapture nonnull readnone %.this_arg) #0 {
  ret i32 0
}

; Function Attrs: uwtable
define i32 @_D5dconf7callFooFCQq1AZi(%dconf.A* %a_arg) local_unnamed_addr #2 {
  %1 = getelementptr inbounds %dconf.A, %dconf.A* %a_arg, i64 0, i32 0
  %2 = load [6 x i8*]*, [6 x i8*]** %1, align 8
  %"a.foo@vtbl" = getelementptr inbounds [6 x i8*], [6 x i8*]* %2, i64 0, i64 5
  %3 = bitcast i8** %"a.foo@vtbl" to i32 (%dconf.A*)**
  %4 = load i32 (%dconf.A*)*, i32 (%dconf.A*)** %3, align 8
  %5 = tail call i32 %4(%dconf.A* nonnull %a_arg)
  %6 = mul i32 %5, 123
  ret i32 %6
}
```

Semantics and magic

- Clear and detailed definition of semantics is paramount
- Semantics must abstract over hardware
  - a “function call” is not necessarily a CPU call instruction (otherwise inlining is impossible)
  - the word “stack” in the spec does not mean the CPU stack (some architectures don't even have stack instructions) (more on this later)
- Optimization and instrumentation depend on these abstract semantics

`immutable`

Profile-Guided Optimization (PGO)

- PGO illustrates LLVM's optimization and instrumentation functionality
- Optimization using two compile steps
  - Compile with instrumentation: `-fprofile-instr-generate`
  - Run program to obtain profile
  - Compile and optimize using profile: `-fprofile-instr-use=<profile>`
- What information should a profile contain?
  - Inlining (or not) plays major role in optimization
    - After inlining, many new optimizations can be performed
  - What kind of information is useful for inlining?
    - control flow (how often is a statement executed)
    - reoccuring values (mainly function pointers → indirect call promotion)

PGO with LLVM in LDC

- LLVM provides:
  - profile file handling (storing/loading/merging/...)
  - intrinsic functions + codegen + runtime library
  - optimizations based on control flow and indirect call pointer value metadata on IR
- LDC must do:
  - add instrumentation code (calls to LLVM intrinsics)
  - calculate information from obtained profile data and add metadata on IR

PGO: control flow instrumentation

```
int bar(bool zero) {
    if (zero)
        return 0;
    else
        return someFunction();
}
```

```
; Function Attrs: norecurse uwtable
define i32 @bar(i1 zeroext %zero_arg) #0 {
  %pgocount = load i64, i64* getelementptr inbounds
      ([2 x i64], [2 x i64]* @__profc_bar, i64 0, i64 0), align 8
  %1 = add i64 %pgocount, 1
  store i64 %1, i64* getelementptr inbounds ([2 x i64], [2 x i64]* @__profc_bar, i64 0, i64 0), a

  br i1 %zero_arg, label %if, label %else

if:                                               ; preds = %0
  %pgocount2 = load i64, i64* getelementptr inbounds
      ([2 x i64], [2 x i64]* @__profc_bar, i64 0, i64 1), align 8
  %2 = add i64 %pgocount2, 1
  store i64 %2, i64* getelementptr inbounds ([2 x i64], [2 x i64]* @__profc_bar, i64 0, i64 1), a
  ret i32 0

else:                                             ; preds = %0
  %3 = tail call i32 @someFunction() #6
  ret i32 %3
}
```

PGO: Indirect Call Promotion (ICP)

```d
int foo()
{
    return 0;
}

int callIndirect(int function() fptr)
{
    return fptr() * 123;
}
```

PGO ICP: instrumentation

```c
int foo()
{
    return 0;
}

int callIndirect(int function() fptr)
{
    return fptr() * 123;
}
```

```
; Function Attrs: norecurse nounwind uwtable
define i32 @foo() #0 {
  %pgocount = load i64, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_foo, i64 0, i
  %1 = add i64 %pgocount, 1
  store i64 %1, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_foo, i64 0, i64 0), a
  ret i32 0
}

; Function Attrs: uwtable
define i32 @callIndirect(i32 ()* %fptr_arg) #1 {
  %pgocount = load i64, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_callIndirect,
  %1 = add i64 %pgocount, 1
  store i64 %1, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_callIndirect, i64 0,

  %2 = ptrtoint i32 ()* %fptr_arg to i64
  tail call void @__llvm_profile_instrument_target(i64 %2, i8* bitcast ({ i64, i64, i64*, i8*, i8

  %3 = tail call i32 %fptr_arg()
  %4 = mul i32 %3, 123
  ret i32 %4
}
```

PGO ICP: optimization with profile

```
; Function Attrs: norecurse nounwind readnone uwtable
define i32 @foo() #0 !prof !28 {
  ret i32 0
}

; Function Attrs: uwtable
define i32 @callIndirect(i32 ()* %fptr_arg) local_unnamed_addr #1 !prof !28 {
  %1 = icmp eq i32 ()* %fptr_arg, @foo
  br i1 %1, label %if.end.icp, label %if.false.orig_indirect, !prof !29

if.false.orig_indirect:                           ; preds = %0
  %2 = tail call i32 %fptr_arg()
  %phitmp = mul i32 %2, 123
  br label %if.end.icp

if.end.icp:                                       ; preds = %0, %if.false.orig_indirect
  %3 = phi i32 [ %phitmp, %if.false.orig_indirect ], [ 0, %0 ]
  ret i32 %3
}

!28 = !{!"function_entry_count", i64 2000}
!29 = !{!"branch_weights", i32 2000, i32 0}
```

More on PGO...

- https://johanengelen.github.io
- Interplay with LTO:
  Jon Degenhardt's DConf 2018 talk
  “Exploring D via Benchmarking of eBay's TSV Utilities”
- Note: D AST-based (this talk)
  versus IR-based (actively developed)

Fuzzing

```
int fuzzMe(in ubyte[] data)
{
    // Test that the first and Nth element are '<' and '>',
    // and that two chars in the middle are equal.
    enum N = 10;
    if (data.length >= N  &&
        data[0] == '<'    &&
        data[N/2] == data[N/2+1] &&
        data[N] == '>')
    {
        return 1;
    }

    return 0;
}
```

normal compilation with LDC:

```
define i32 @_D10dconf_fuzz6fuzzMeFxAhZi({ i64, i8* } %data_arg) local_unnamed_addr #0 {
  %data_arg.fca.0.extract = extractvalue { i64, i8* } %data_arg, 0
  %data_arg.fca.1.extract = extractvalue { i64, i8* } %data_arg, 1
  %1 = icmp ugt i64 %data_arg.fca.0.extract, 9
  br i1 %1, label %bounds.ok, label %andandend16

bounds.ok:                                        ; preds = %0
  %2 = load i8, i8* %data_arg.fca.1.extract, align 1
  %3 = icmp eq i8 %2, 60
  br i1 %3, label %bounds.ok11, label %andandend16

bounds.ok11:                                      ; preds = %bounds.ok
  %4 = getelementptr i8, i8* %data_arg.fca.1.extract, i64 5
  %5 = load i8, i8* %4, align 1
  %6 = getelementptr i8, i8* %data_arg.fca.1.extract, i64 6
  %7 = load i8, i8* %6, align 1
  %8 = icmp eq i8 %5, %7
  br i1 %8, label %andand15, label %andandend16

andand15:                                         ; preds = %bounds.ok11
  %bounds.cmp18 = icmp ugt i64 %data_arg.fca.0.extract, 10
  br i1 %bounds.cmp18, label %bounds.ok19, label %bounds.fail20

bounds.ok19:                                      ; preds = %andand15
  %9 = getelementptr i8, i8* %data_arg.fca.1.extract, i64 10
  %10 = load i8, i8* %9, align 1
  %11 = icmp eq i8 %10, 62
  %phitmp = zext i1 %11 to i32
  br label %andandend16
```

with LDC flag: `-fsanitize=fuzzer`

```
define i32 @_D10dconf_fuzz6fuzzMeFxAhZi({ i64, i8* } %data_arg) local_unnamed_addr #0 {
  call void @__sanitizer_cov_trace_pc_guard(i32* getelementptr inbounds ([5 x i32], [5 x i32]* @_
  call void asm sideeffect "", ""()
  %data_arg.fca.0.extract = extractvalue { i64, i8* } %data_arg, 0
  %data_arg.fca.1.extract = extractvalue { i64, i8* } %data_arg, 1
  call void @__sanitizer_cov_trace_const_cmp8(i64 9, i64 %data_arg.fca.0.extract)
  %1 = icmp ugt i64 %data_arg.fca.0.extract, 9
  br i1 %1, label %bounds.ok, label %.andandend16_crit_edge

.andandend16_crit_edge:                           ; preds = %0
  call void @__sanitizer_cov_trace_pc_guard(i32* inttoptr (i64 add (i64 ptrtoint ([5 x i32]* @__s
  call void asm sideeffect "", ""()
  br label %andandend16

bounds.ok:                                        ; preds = %0
  %2 = load i8, i8* %data_arg.fca.1.extract, align 1
  call void @__sanitizer_cov_trace_const_cmp1(i8 60, i8 %2)
  %3 = icmp eq i8 %2, 60
  br i1 %3, label %bounds.ok11, label %bounds.ok.andandend16_crit_edge

bounds.ok.andandend16_crit_edge:                  ; preds = %bounds.ok
  call void @__sanitizer_cov_trace_pc_guard(i32* inttoptr (i64 add (i64 ptrtoint ([5 x i32]* @__s
  call void asm sideeffect "", ""()
  br label %andandend16

bounds.ok11:                                      ; preds = %bounds.ok
  %4 = getelementptr i8, i8* %data_arg.fca.1.extract, i64 5
  %5 = load i8, i8* %4, align 1
  %6 = getelementptr i8, i8* %data_arg.fca.1.extract, i64 6
  %7 = load i8, i8* %6, align 1
  call void @__sanitizer_cov_trace_cmp1(i8 %5, i8 %7)
  %8 = icmp eq i8 %5, %7
```

Fuzzing + extra sanity checks

- LLVM provides a memory safety checker: Address Sanitizer (ASan)
- ASan is a combination of
  - compiler-inserted instrumentation
  - runtime library to manage memory and keep track of valid and invalid memory locations (“poisoning”)
- LDC flag: `-fsanitize=address`

ASan and “stack”

```
 1  class A {
 2      int i;
 3  }
 4
 5  void inc(A a) {
 6      a.i += 1; // Line 6
 7  }
 8
 9  auto makeA() {  // Line 9
10      import std.algorithm : move;
11      // "scope" allocates object on the stack instead of the heap
12      scope a = new A();
13      return move(a);
14  }
15
16  void main() {
17      auto a = makeA();
18      a.inc(); // Line 18
19  }
```

More on libFuzzer and ASan...

- libFuzzer documentation
  https://llvm.org/docs/LibFuzzer.html
- libFuzzer tutorial (interested in discovering the Heartbleed bug?)
  https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md
- Address Sanitizer documentation
  https://github.com/google/sanitizers/wiki/AddressSanitizer
- “Finding memory bugs in D code with AddressSanitizer” and “Fuzzing D code with LDC”
  https://johanengelen.github.io

One last demo: let's fuzz Jonathan's `dxml` library?

```d
import ldc.libfuzzer;
mixin DefineTestOneInput!fuzzMe;

int fuzzMe(in ubyte[] data)
{
    import dxml.parser;
    try
    {
        int sum;
        auto range = parseXML(cast(char[])data);
        foreach (elem; range) {
            // Do something unpredictable to actually test the parser
            sum += elem.name.length;
        }
        return sum > 1;
    }
    catch (XMLParsingException)
    {
        return 0;
    }
}
```

Summary

- *If you want to learn about low-level details of compilation:*
  Read LLVM's mailing list
- *If you want your code to run fast:*
  Use PGO and, more important, LTO!
- *If you want your code to run safely:*
  Start fuzzing with ASan enabled
- *If you are interested in working on LDC:*
  Good first PR: optimizing Walter's `if(0){ … }` trick that Eduard presented yesterday. (not trivial!)
- *If you want to join me in working on aggressive optimizations:*
  Let's add those nitty gritty details to the spec

Just eliminate `if(0){...}`?

```
if (a) goto lab1;
if (0) {
    lab1:
    // ...
}
```