LDC: The LLVM-based D Compiler
Using LLVM as backend for a D compiler

Kai Nacke
02/02/14
LLVM devroom @ FOSDEM´14
Agenda

• Brief introduction to D

• Internals of the LDC compiler

• Used LLVM features

• Possible improvements of LLVM
What is D?

- C-like syntax
- Static typing
- Supports many paradigms
  - Polymorphism, functional style, generics, contract programming
- Scales up to large projects
  - Modules, interfaces, unit tests
- Convenient and powerful features
  - Garbage collection, array slices, compile-time function execution (CTFE)
```cpp
void main() {
    import std.stdio;
    writeln("Hello FOSDEM!");
}

module myalgo;
import std.traits;
T gcd(T)(T a, T b) pure
    if(isIntegral!T) {
        while (b) {
            auto t = b;
            b = a % b;
            a = t;
        }
        return a;
    }
unittest {
    // CTFE
    enum val1 = gcd(3, 5);
    assert(val1 == 1);
    // No CTFE
    assert(gcd(25, 35) == 5);
}
```

```cpp
import std.stdio;
import std.array;
import std.algorithm;

void main() {
    stdin.byLine(KeepTerminator.yes). map!(a => a.idup). array. sort. copy(stdout.lockingTextWriter());
}
```
void main() {
    import std.stdio;
    writeln("Hello FOSDEM!");
}

module myalgo;
import std.traits;
T gcd(T)(T a, T b) pure nothrow
if(isIntegral!T) {
    while (b) {
        auto t = b;
        b = a % b;
        a = t;
    }
    return a;
}

unittest {
    // CTFE
    enum val1 = gcd(3, 5);
    assert(val1 == 1);
    // No CTFE
    assert(gcd(25, 35) == 5);
}

import std.stdio;
import std.array;
import std.algorithm;
void main() {
    stdin.byLine(KeepTerminator.yes).map!(a => a.idup).array.sort.copy(stdout.lockingTextWriter());
}
Code Examples - gcd

```plaintext
module myalgo;

import std.traits;

T gcd(T)(T a, T b) pure if(isIntegral!T) {
    while (b) {
        auto t = b;
        b = a % b;
        a = t;
    }
    return a;
}

unittest {
    // CTFE
    enum val1 = gcd(3, 5);
    assert(val1 == 1); // No CTFE
    assert(gcd(25, 35) == 5);
    // No CTFE
    assert(gcd(25, 35) == 5);
}
```
import std.stdio;
import std.array;
import std.algorithm;

void main() {
    import std.stdio;
    writeln("Hello FOSDEM!");
}

module myalgo; import std.traits;
T gcd(T)(T a, T b) pure nothrow
if(isIntegral!T) {
    while (b) {
        auto t = b; b = a % b; a = t;
    }
    return a;
}

unittest {
    // CTFE
    enum val1 = gcd(3, 5); assert(val1 == 1); // No CTFE
    assert(gcd(25, 35) == 5);
}

import std.stdio; import std.array; import std.algorithm; void main() {
    import std.stdio;
    stdin.byLine(KeepTerminator.yes). map!(a => a.idup). array. sort. copy(stdout.lockingTextWriter());
}
The case for D

• 18th place on the TIOBE index
• Companies adopting D
• Books about D
What about combining the D frontend with LLVM? The idea is about 10 years old! The result is LDC.
Facts about LDC

• Version 0.13.0 alpha recently announced
• Written in C++ (transition to D planned)
• Requires LLVM 3.1 or later
• Runs on most Posix-like x86/x86_64 OS’s  
  – Linux, OS X, FreeBSD, Mingw32
• Native Windows version depends on LLVM
• Port to Linux/PPC64 almost finished
• Work on port to Linux/ARM has started
The architecture of LDC

- Typical multi-pass compiler
  - Lexer, Parser, Analyzer from DMD
  - Type mapping and IR generation
  - Code generation with LLVM
- Illustrates Conway’s law
Closer look at data flow

Reads source files

Creates Abstract Syntax Tree (AST)

Lexer, Parser, Analyzer

Type mapping IR generation

LLVM

LLVM API

.a

.a

.a

.so

.a

.so

.a

.s

.ill

With gcc/clang

Driver

With gcc/clang

.a.out

2/2/2014

11/36
Abstract Syntax Tree from Frontend

- Frontend generates fully decorated AST

```c
if (b > 0) {
  writefln("b > 0");
} else
  writefln("b <= 0");
```

- IfStatement
  - Condition : CmpExpr
  - Ifbody : ScopeStatement
  - Elsebody : ExpStatement
• Frontend already lowers some statements

  ```
  scope(exit) { ... } /* stmts */
  ```

  ```
  try { /* stmts */ } finally { ... }
  ```

  ```
  foreach (a; 1..10) {
    /* stmts */
  }
  ```

  ```
  for (int key = 1;
       key < 10; ++key) {
    /* stmts */
  }
  ```

• Simplifies IR generation

• Complicates generation of debug info
IR generation

• IR is generated with a tree traversal

• Uses visitor provided by frontend
  – New code, it is different in current release

• IR is attached to main entities
  – Example: FuncDeclaration <- IrFunction

• Generated IR could be improved
IR generation example - if

```c
int a;
if (a > 0)
    ...
else
    ...
```

```assembly
%tmp = load i32* %a
%tmp1 = icmp sgt i32 %tmp, 0
br il %tmp1, label %if,
    label %else

if:
    ...
    br label %endif

else:
    ...
    br label %endif

endif:
```
IR generation example - for

```java
for (int i = 0; i < a.length; ++i)
{
    ...
}
```

```assembly
...  
br label %forcond  
forcond:  
    ...  
br i1 %tmp6,  
    label %forbody,  
    label %endfor  
forbody:  
    ...  
br label %forinc  
forinc:  
    ...  
br label %forcond  
endfor:
```
Type mapping

- **Mapping of simple types**
  
<table>
<thead>
<tr>
<th>Simple Type</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>ubyte</td>
</tr>
<tr>
<td>short</td>
<td>ushort</td>
</tr>
<tr>
<td>int</td>
<td>uint</td>
</tr>
<tr>
<td>long</td>
<td>ulong</td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td></td>
</tr>
<tr>
<td>i8</td>
<td>i16</td>
</tr>
<tr>
<td>i32</td>
<td>i64</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>x86_fp80</td>
<td>ppc_fp80</td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

- **Mapping of type bool**

- **Use of i1 for bool turned out to be wrong!**
Type mapping

- Static arrays are mapped to LLVM arrays
  \[ \text{int}[5] \quad \rightarrow \quad [5 \times \text{i32}] \]

- Dynamic arrays are mapped as anonymous structs with length and pointer to data
  \[ \text{int}[\] \quad \rightarrow \quad \{ \text{i64}, \text{i32}* \} \]

- Associative arrays are opaque to the compiler
  \[ \text{int}[\text{string}] \quad \rightarrow \quad \text{i8}* \]
Type mapping

• Structs are mapped to LLVM structs

```c
struct {
    byte a;
    byte b;
    short c;
}
```

```c
type {
    i8, i8, i16
}
```

• Classes are structs with a vtable and a monitor

```c
class {
    long a;
}
```

```c
type {
    %type.ClassA.__vtbl*, i8*, i64
}
```

• Padding is added if required
D-specific LLVM passes

• LLVM knows nothing about D and its runtime library!

• Adding D-specific knowledge to LLVM helps to optimize the code

• Currently these passes are available
  – GarbageCollect2Stack
  – SimplifyDRuntimeCalls
  – StripExternals
D-specific LLVM passes

• GarbageCollect2Stack
  – Tries to turn a GC allocation into a stack allocation
  – Useful for closures

```c
int b = 5;
foreach (int i; 0..10)
{
    apply(a => b*a, i);
    b = 2*b;
}
```

• Requires memory allocation for nested frame
• Can be turned into alloca if memory does not escape function
• Based on PointerMayBeCaptured
• Conservative in loops, can be improved
D-specific LLVM passes

- SimplifyDRuntimeCalls
  - Replaces/optimizes calls of D runtime, mainly for arrays
  - Framework copied from SimplifyLibcalls

- StripExternals
  - Removes body of functions declared as available_externally
  - Used as support for global dead code elimination (GlobalDCE)
Porting to new platforms

• Required LLVM features
  – Thread-local storage (TLS)
  – Exception handling
  – Anonymous structs

• Nice to have
  – Inline assembler
  – Debug symbols
  – Some intrinsics

• Features are not supported by all LLVM targets
Porting to new platforms

• Common problems found in LLVM
  – TLS is not implemented / partially implemented / buggy
  – Wrong relocation generated
    • Most of the time TLS relocations

• E.g. on Linux/ARM – missing R_ARM_TLS_LDO32

```bash
localhost tmp # /build/work/ldc/bin/ldc2 -g hello.d /usr/lib/gcc/armv7a-hardfloat-linux-gnueabi/4.8.2/.../../.../../../armv7a-hardfloat-linux-gnueabi/bin/ld: /build/work/ldc/runtime/../../lib/libphobos-ldc-debug.a(random-debug.o)(.debug_info+0x31): R_ARM_ABS32 used with TLS symbol _D3std6random17unpredictableSeedFNdZk6seededdb
```
Porting to new platforms

• Recommendation for porting

  Always use LLVM trunk!

• A new port can improve quality of LLVM
  – E.g. PowerPC64: 17 bug reports

• Many interesting platforms are yet unsupported (MIPS, AArch64, Sparc, ...)

2/2/2014
Inline ASM

- LDC supports DMD-style ASM on x86/x86_64

```plaintext
asm {
  naked;
  mov RAX, 8;
  mov RAX, GS:[RAX];
  ret;
}
```

- Inline assembler requires parsing of statements and construction of constraints
- Naked functions are translated to modul-level inline assembly
- Is not inlined by default
Inline ASM

- LLVM ASM is supported on all platforms via a special template

```python
import ldc.llvmasm;

return __asm!(void*)("movq %gs:8, %rax", "={rax}");
```

- Preferred way because it can be inlined
- No debug info generated
Inline IR

- IR can be inlined via a special template

```c
#pragma(LDC_inline_ir)
R inlineIR(string s, R, P...)(P);

void* getStackBottom(){
    return inlineIR!(`%
        %ptr = inttoptr i64 %0 to i64 addrspace(256)*
        %val = load i64 addrspace(256)* %ptr, align 1
        %tmp = inttoptr i64 %val to i8*
        ret i8* %tmp`,
    void*, ulong)(8);
}
```

- Be aware that IR is not platform independent!
Inline IR

• Example translates to (EH removed)

```
_D3thr14getStackBottomFZPv:
  movq  %gs:8, %rax
  retq
```

• Useful to access IR features which are otherwise not available (e.g. shufflevector)

• Best result because of tight integration with LLVM

• No debug info generated
Attributes

• Used to change LLVM function attributes

```java
import ldc.attribute;

@attribute("alwaysinline")
void func() {
    // ...
}
```

• Some attributes require more work
• Experimental feature, not yet finished
Integrating AdressSanitizer

• Integrate sanitizer passes into compile
  – opt is used as blueprint

• Add new option
  – --sanitize=address

• Add attribute SanitizeAddress to every function definition
Integrating AdressSanitizer

• Compile runtime library with new option

• Still missing: runtime support
  – Own allocator with GC
  – Use gcstub/gc.d instead

• Produces some aborts in unit tests on Linux/x86_64

• Evaluation of reports not complete
Better ABI support

• Frontend must have intimate knowledge of calling convention

• Degree of implementation of attributes varies
  – PPC64: Good
  – Win64: Needs more work (byval with structs)

• LDC uses an abi class to encapsulate the details for each supported platform

• Improvement: more complete implementation
Better ABI support

- Improvement: Implement helper for C ABI lowering in LLVM

- It is the default ABI in LLVM and every major language needs this

- Think of the effort required for MSC ABI in Clang

- Even better: Abstract the details away
Better Windows support

• No exception handling on native Windows
  – Most wanted feature by LDC users!
  – I am working on the implementation
  – There is already a Clang driver in PR18654

• No CodeView debug symbols
  – COFF line number support added recently
  – I started to work on this topic (a year ago...)

2/2/2014 35 / 36
Looking for contributors

If you want to have fun with
  ... a cool language
  ... a friendly community
  ... hacking LLVM

then start contributing to LDC today!

http://forum.dlang.org/group/digitalmars.D.ldc
https://wiki.dlang.org/LDC
https://github.com/ldc-developers