LLVM Code Generation for Open Dylan

Peter S. Housel
April 27, 2020
Introduction
The Dylan Programming Language

- Originated at the Apple Advanced Technology Group in the early 1990s
  - Initially targeting the Apple Newton PDA as a systems language
  - Later promoted as an applications language for the classic Macintosh
- Carnegie Mellon University Gwydion Dylan project (\texttt{d2c} compiler), later maintained by our group (1998-2011)
- Harlequin Dylan
  - Later spun off as Functional Developer by Functional Objects
  - Open sourced as Open Dylan in 2004
  - Includes DUIM (successor of CLIM), IDE, debugger, database interfaces, …
• Designed as an *application delivery* language

• A “**Dynamic Language**” (compared to 1990s C++ or Object Pascal), but with features designed to enable efficient compiled code
  
  • Library-at-a-time compilation
  
  • *Sealing* of classes or generic functions, allowing type inference, method inlining, or specific method dispatch.

```dylan
define sealed domain make (singleton(<standard-display>));
define sealed domain initialize (<standard-display>);
```
**Dylan Flow Machine Compiler Structure**

**Figure 1: DFMC Compiler Structure**
The LLVM Back-End
LLVM Back-End Goals

1. Support debug information (DWARF)
2. Expand code generation support to other architectures (x86_64, AArch64)
3. Avoid inefficiencies incurred by compiling via C code.
4. Take advantage of optimizations provided by the LLVM compiler infrastructure.
5. Support integration with non-conservative garbage collectors such as the Memory Pool System.
The LLVM Intermediate Representation language:

- Single-Static Assignment (SSA) representation
- Representation used for most optimizations
- Input to machine code generation

```c
define fastcc %struct.dylan_mv_ @KemptyQVKdMM10I(i8* %listF1, i8* %.next, i8* %.function) {
  bb.entry:
    %0 = icmp eq i8* %listF1, bitcast (%KLempty_listGVKd* @KPempty_listVKi to i8*)
    %1 = select i1 %0, i8* bitcast (%KLbooleanGVKd* @KPtrueVKi to i8*), i8* bitcast (%KLbooleanGVKd* @KPfalseVKi to i8*)
    %2 = insertvalue %struct.dylan_mv_ undef, i8* %1, 0
    %3 = insertvalue %struct.dylan_mv_ %2, i8 1, 1
    ret %struct.dylan_mv_ %3
}
```
Approaches to generating LLVM IR:

- Linking with and calling the LLVM libraries
  - Requires C-FFI interface to LLVM C interface
  - Requires linking with large shared library

- Writing textual LLVM assembly language
  - Can be straightforward to output from a native IR representation
  - Greater I/O overhead
  - Fewer forward-compatibility guarantees

- Writing out LLVM bitcode
  - Nontrivial to implement
  - Best level of forward compatibility
• LLVM constant and instruction values are explicitly typed
• Heap objects

\%KLmm\_wrapperGVKi = type \{ \%KLmm\_wrapperGVKi*, i8*, i8*, i64, i64, i8*, [0 x i64] \}
\%KLlistGVKd = type \{ \%KLmm\_wrapperGVKi*, i8*, i8* \}

• Tagged pointers

\%37 = ptrtoint i8* \%remainingF39 to i64
\%38 = and i64 \%37, 3
switch i64 \%38, label \%48 [
  i64 0, label \%39
]
define sealed inline method \+
   (x :: <single-float>, y :: <single-float>)
=> (z :: <single-float>)
   primitive-raw-as-single-float
   (primitive-single-float-add
    (primitive-single-float-as-raw(x),
     primitive-single-float-as-raw(y)))
end method;
define fastcc %struct.dylan_mv_ @KAVKdMM2I(i8* %xF1, i8* %yF2, i8* %.next, i8* %.function) {
    bb.entry:
    %0 = bitcast i8* %xF1 to %KLsingle_floatGVKd*
    %1 = getelementptr inbounds %KLsingle_floatGVKd, %KLsingle_floatGVKd* %0, i64 0, i32 1
    %2 = load float, float* %1, align 8
    %3 = bitcast i8* %yF2 to %KLsingle_floatGVKd*
    %4 = getelementptr inbounds %KLsingle_floatGVKd, %KLsingle_floatGVKd* %3, i64 0, i32 1
    %5 = load float, float* %4, align 8
    %6 = fadd float %2, %5
    %7 = call fastcc %KLsingle_floatGVKd* @primitive_raw_as_single_float(float %6)
    %8 = bitcast %KLsingle_floatGVKd* %7 to i8*
    %9 = insertvalue %struct.dylan_mv_ undef, i8* %8, 0
    %10 = insertvalue %struct.dylan_mv_ undef, i8 1, 1
    ret %struct.dylan_mv_ %10
}
Define side-effect-free stateless dynamic-extent
&runtime-primitive-descriptor primitive-wrap-unsigned-abstract-integer
  (x :: <raw-machine-word>) => (result :: <abstract-integer>);

let word-bits = back-end-word-size(be) * 8;

let maximum-fixed-integer
  = generic/-(generic/ash(1, word-bits - $dylan-tag-bits - 1), 1);

// Check for greater than maximum-fixed-integer
let cmp-above = ins--icmp-ugt(be, x, maximum-fixed-integer);

ins--if (be, cmp-above)
  // Allocate and initialize a <double-integer> instance
  let class :: <&class> = dylan-value(#"<double-integer>";
  let double-integer = op--allocate-untraced(be, class);
  let low-slot-ptr
    = op--getslotptr(be, double-integer, class, #"%%double-integer-low");
  ins--store(be, x, low-slot-ptr);
  let high-slot-ptr
    = op--getslotptr(be, double-integer, class, #"%%double-integer-high");
  ins--store(be, 0, high-slot-ptr);
  ins--bitcast(be, double-integer, $llvm-object-pointer-type)
ins--else
  // Tag as a fixed integer
  let shifted = ins--shl(be, integer-value, $dylan-tag-bits);
  let tagged = ins--or(be, shifted, $dylan-tag-integer);
  ins--inttoptr(be, tagged, $llvm-object-pointer-type)
end ins--if;
end;
IEP  Internal Entry Points

• Arity known, keyword arguments split
• Artificial .next (used for next-method dispatch) and .function (used for accessing closed-over values) arguments passed at the end of the argument list
• fastcc LLVM calling convention

```c
define fastcc %struct.dylan_mv_ @Ktype_check_errorVKiI(i8* %valueF1,
   i8* %typeF2,
   i8* %.next,
   i8* %.function) {

; ... 
}
```
**XEP** External Entry Points

- Arity unknown to caller
- ccc LLVM calling convention, possibly with varargs

```c
define %struct.dylan_mv_ @xep_1(i8* %function, i64 %n, i8* %a2) {
    ; ...
}
```
Engine Node  Dispatch Engine Node Entry Points

- Used to evaluate method dispatch decision tree steps (or chain to Dylan code that does)
- ccc LLVM calling convention

```c
define %struct.dylan_mv_ @if_type_discriminator_0_1(i8* %engine,
  i8* %function,
  i8* %a2) {

  bb.entry:
  ; ...
}
```
**MEP** Method Entry Points

- Does keyword argument and `#rest` processing and chains to the IEP
- ccc LLVM calling convention, with varargs

```llvm
define %struct.dylan_mv_ @rest_key_mep_1(i8* %meth, i8* %next_methods, ...
    ; ...
}
```
Multiple Return Values

- Vector of 64 return values in thread-local storage

```c
struct dylan_teb {  // Thread Environment Block
    D teb_dynamic_environment;
    D teb_thread_local_variables;
    D teb_current_thread;
    D teb_current_thread_handle;
    D teb_current_handler;
    D teb_runtime_state;
    D teb_pad[2];
    D teb_mv_count;
    D teb_mv_area[64];
};
```
IEPs and entry points return the primary value and return value count, as a `struct` return (two registers for most ABIs)

```plaintext
%struct.dylan_mv_ = type { i8*, i8 }
```

Within functions, multiple returned values are treated as local SSA values (registers and stack) whenever possible.
Foreign Function Interface

• Interoperation with C (and Objective C) using raw types
• Takes advantage of built-in LLVM support for these calling conventions
• Challenge of `struct/array` call and return (only minimally modeled by LLVM)
• Dylan block construct

```dylan
define method get-file-property
  (pathname :: <pathname>, property, #key default = $unsupplied) => (value)
  if (unsupplied?(default))
    file-property(pathname, property)
  else
    block ()
      let value = file-property(pathname, property);
      value
    exception (<condition>)
      default // if there's an error, return the default
  end
end
end method get-file-property;
```
define fastcc %struct.dylan_mv_ @Kget_file_propertyYdeuce_internalsVdeuceMM0I
  (i8* %pathnameF1, i8* %propertyF2, i8* %UrestF3, i8* %defaultF4,
   i8* %next, i8* %function)
  personality i32 (...)* @__opendylan_personality_v0 !dbg !80 {
    bb.entry:
    ; ...
    %79 = invoke %struct.dylan_mv_ %78
       (i8* bitcast (%KLsealed_generic_functionGVKe* @Kfile_propertyYfile_systemVsystem to i8*),
        i64 2, i8* %pathnameF1, i8* %propertyF2)
       to label %80 unwind label %81, !dbg !100

    ; ...
    %81 = landingpad { i8*, i32 }
    ; preds = %74
    cleanup
    catch i8** @Kget_file_propertyYdeuce_internalsVdeuceMM0I.Uunwind_exceptionUPIexit_3F12, !dbg !103
    ; ...
  }
• Low overhead in the usual case
• Explicit compilation model of nonlocal control flow
• If nonlocal exits are frequent, `libunwind` and the system dynamic library loader have a high run-time cost
• Open Dylan supports thread-local variable definitions
  
  ```dylan
define thread variable *jam-input-state* :: <jam-input-state>
  = make(<jam-input-state>, input-data: "")
  ```

• LLVM has direct support for thread-local variables
  
  ```c
  @Tjam_input_stateTYjam_internalsVjam
  = thread_local global i8* bitcast (%KLunboundGVKe* @KPunboundVKi to i8*),
    align 8
  ```
Thread-Local Storage

- Challenge of ensuring that variables are initialized in new threads, especially when libraries can be loaded dynamically

```llvm
%117 = load i64, i64* @Ptlv_initializations_cursor, align 8
%118 = load i64, i64* @Ptlv_initializations_local_cursor, align 8
%119 = icmp ult i64 %118, %117
%120 = call i1 @llvm.expect.i1(i1 %119, i1 false) #3
br i1 %119, label %121, label %122

121:
call void @primitive_initialize_thread_variables()
br label %122

122:
; load from @Tjam_input_stateTYjam_internalsVjam
```
• LLVM functions and instructions can be annotated with debugging metadata, translated by the code generator to DWARF or Microsoft CodeView format.

• Basic source location and local variable information for Dylan programs works in LLDB, the LLVM project debugger.
• Work is in progress to integrate the Open Dylan debugger (supporting breakpoints, stepping, local variable access, and an interactive REPL) with LLDB using remote procedure call
Build System Integration
• Open Dylan development environment needs to call on Clang and the system linker to build applications and shared libraries

• Need to support a variety of external toolchains on Windows, Linux, BSD, and macOS platforms

• Our solution uses an interpreted Domain-Specific Language based on the Jam build system
  • Language defines build steps, build targets, and their dependencies
  • Build execution engine performs parallel execution of the build toolchain
Conclusion
Conclusion

- **Website:** http://opendylan.org/
- **Dylan-Lang Community:** https://gitter.im/dylan-lang/general