Generating LLVM IR Code from OCaml

Letterio Galletta

Agenda

- LLVM API in OCaml
  - Module declaration
  - Function declaration
  - Statement and expressions
- Code generation for first order FUN

References

- OCaml bindings documentation
- Kaleidoscope: Implementing a Language with LLVM in Objective Caml

LLVM: OCaml binding

LLVM Core Libraries are written in C++, but OCaml binding are available on opam

```
$ opam install llvm
```

The api documentation is available at

https://llvm.moe/ocaml/

To compile a file.ml use

```
$ ocamlbuild -use-ocamlfind -package llvm  file.byte
```

What is in API?

- The OCaml binding provides you a set of values, functions, and types to programmatically creates LLVM IR code
- Each construct of LLVM IR is represented by a value of a suitable type (see next slide)
- The generation of LLVM IR code reduces to create the corresponding OCaml values

Main types

We will create mainly values of the following types

- llcontext: the top-level container for all LLVM global data
- llmodule: the top-level container for all other LLVM IR objects
- llvalue: any element of the LLVM IR: functions, instructions, global variables, constants, are all llvalues
- lltype: types of LLVMIR, e.g., i32, i1, [i32 x 8], etc.
- llbasicblock: a basic block to build the body of functions
- llbuilder: used to generate instructions in the LLVM IR

For other elements of the IR, e.g., attribute (see documentation)

Code generation workflow

The typical steps for generating LLVM IR code:

1. Get a value representing the llvm context
2. Create a module containing all our code
3. Fill in the module with the declarations of functions and globals
    a. For functions: declare parameters, local variables and blocks
4. Serialize the module either in assembly or in bitcode

LLVM context

A `llcontext` value represents the container that we use to create `llmodules`, `llvalues` and `lltypes`

By default there exists a predefined global context that we can use

To get a the global context you can use

```
val global_context : unit -> llcontext
```

There are also functions to create a new context and dispose it

```
val create_context : unit -> llcontext
val dispose_context : llcontext -> unit
```

LLVM module

A `llmodule` contains global variables, functions, data layouts, host triple, ...

- `create_module : llcontext -> string -> llmodule`

  `create_module context id` creates a module with the supplied module ID in the context `context`. Modules are not garbage collected; it is mandatory to call `Llvm.dispose_module` to free memory

- `dispose_module : llmodule -> unit`

  `dispose_module m` destroys a module `m` and all of the IR objects it contained

There are functions to set data layout, host triples, and to save the module on a file

An example of module

```
let llvm_context = Llvm.global_context () in
let my_module = Llvm.create_module llvm_context "my-empty-module" in
Printf.printf "%s" (Llvm.string_of_llmodule my_module);
Llvm.dispose_module my_module
```

Emit a module in assembly

See the file empty_module.ml

An example of module: bitcode generation

```
let filename = "output.bc" in
let llvm_context = Llvm.global_context () in
let my_module = Llvm.create_module llvm_context "my-empty-module" in
Llvm_bitwriter.write_bitcode_file my_module filename |> ignore;
Llvm.dispose_module my_module
```

See the file empty_module_bitcode.ml

See modules `Llvm_bitwriter` / `Llvm_bitreader` for functions that output/input LLVM IR in bitcode format

Define global variables

We use the following function to define a global variable and initialize it

- `define_global : string -> llvalue -> llmodule -> llvalue`
  `define_global name init m` returns a new global with name `name` and initializer `init` in module `m`

There are also functions to lookup a global variable from the global scope, to delete it and to set different attribute (see documentation)

Example of global variables

See the file global_variables.ml

We create three global variables initialized by a literal

1. In `helloworld_string`, we use `Llvm.const_stringz` to create a null-terminated literal string
2. In `integer_constant` we use `Llvm.i32_type` to create a value to represent the LLVM type `i32`
3. In `array_constant` we use
   - `Llvm.float_type` to create the type for the elements of the array
   - `Llvm.array_type` to create the type of the array
   - `Llvm.const_float` to create each single element of the array
   - We create an (OCaml) array of llvalues to initialize our literal array

Define functions

Two steps are required:

1. Define the type of the function
   ◦ `function_type : lltype -> lltype array -> lltype`

   `function_type ret_ty param_tys` returns the function type returning `ret_ty` and taking `param_tys` as parameters

2. Define the function itself
   - `declare_function : string -> lltype -> llmodule -> llvalue`

   `declare_function name ty m` returns a new function of type `ty` and with name `name` in module `m`

There are also functions to lookup functions from the global scope, to delete it and to set different attribute (see documentation)

Example of function definition

```
let voidt = Llvm.void_type llvm_context  in
let ttmain = Llvm.function_type voidt [||]  in
let fundef = Llvm.define_function  "main" ttmain llmodule  in
...
```

See the file fun_def_void.ml

- We use `Llvm.void_type llvm_context` to create a value representing the type void
- Our function has no arguments (`[||]`) and returns void

Basic blocks

- A function is made of a set of basic blocks that are values of type `llbasicblock`
- When we define a function there is a initial basic block for the entry point

To obtain the entry block of a function we can use

```
entry_block : llvalue -> llbasicblock
```

There are other functions that allow manipulating basic blocks

- `append_block c name f` creates a new basic block named `name` at the end of function `f` in the context `c`
- `delete_block bb` deletes the basic block `bb`
- `move_block_before pos bb` moves the basic block `bb` **before** `pos`

See documentation for other functions on basic blocks

Instruction builders

- A builder is used to construct LLVM IR instruction and it is a value of type `llbuilder`
- A builder has a position that points to the basic block to which we add the new generated instructions
- To create a builder we can use
  - `builder context` creates an instruction builder with no position in the context `context`. It is invalid to use this builder until its position is set with `Llvm.position_before` or `Llvm.position_at_end`.
  - `builder_at_end bb` creates an instruction builder positioned at the end of the basic block `bb`

See documentation for other functions

Generating the body of a function

- To generate the code of a function we need to generate its basic blocks and fill them with instructions
- Typically, we start generating code starting from the entry block of a function, the expression

  ```
  Llvm.builder_at_end llctx (Llvm.entry_block f)
  ```

  returns an instruction builder positioned at the end of the entry of a function

- We can add new basic blocks to a function and use the function in the slide before to make the instruction builder pointing to the correct block

Example of a function body

```
let llvm_context = Llvm.global_context ()  in
...
let fundef = Llvm.define_function  "main" ttmain llmodule  in
let ibuilder = Llvm.builder_at_end  llvm_context (Llvm.entry_block fundef)  in
let _ = Llvm.build_ret_void ibuilder  in
...
```

See fun_body_void.ml

- `Llvm.build_ret_void` returns a “return void” instruction
- The new instruction is inserted at the end of the entry block of the function

Function parameters

- The (types of) parameters of a function are specified in the type of the function
- We can access the parameters of a function by their position using the following functions

`params f` returns the parameters of function `f`

`param f n` returns the `n`th parameter of function `f`

`iter_params f fn` applies function `f` to each of the parameters of function `fn` in order

`fold_left_params f init fn` similar to fold_left on List but over the list of params

Example of a function with parameters

```
let llvm_context = Llvm.global_context ()  in
let llmodule = Llvm.create_module llvm_context  "my-empty-module"  in
let i32t = Llvm.i32_type llvm_context  in
let ttsum = Llvm.function_type i32t [|i32t ; i32t|]  in
let fundef = Llvm.define_function  "sum" ttsum llmodule  in
…
```

See fun_params.ml

Arithmetic/bitwise expressions

There are functions with name `build_*` that allow to build arithmetic instructions

- `build_add x y name b` creates a `%name = add %x, %y` instruction at the position specified by the instruction builder `b`
- `build_fadd x y name b` creates a `%name = fadd %x, %y` instruction at the position specified by the instruction builder `b`
- `build_shl x y name b` creates a `%name = shl %x, %y` instruction at the position specified by the instruction builder `b`

There are many functions available see the documentation

Example of arithmetic expressions

```
let param0 = Llvm.param fundef 0  in

let param1 = Llvm.param fundef 1  in

let sum_param = Llvm.build_add param0 param1  "result" ibuilder in

let _ = Llvm.build_ret sum_param ibuilder
```

See the file fun_sum.ml

- We get a reference to a parameter of the function using `Llvm.param`
- We return the sum of the two parameters

Manipulate memory

To build alloca, load and store instructions we can use following `build_*` functions

- `build_alloca ty name b` creates a `%name = alloca %ty` instruction at the position specified by the instruction builder `b`
- `build_array_alloca ty n name b` creates a `%name = alloca %ty, %n` instruction at the position specified by the instruction builder `b`
- `build_load v name b` creates a `%name = load %v` instruction at the position specified by the instruction builder `b`
- `build_store v p b` creates a `store %v, %p` instruction at the position specified by the instruction builder `b`

Example of local variables (1)

```
let param_stack = Llvm.build_alloca i32t "param" ibuilder in

let param = Llvm.param fundef 0 in

let _ = Llvm.build_store param param_stack ibuilderin

let load_param = Llvm.build_load param_stack "load" ibuilder in

let sum_param = Llvm.build_add load_param (Llvm.const_int i32t 2) "result" ibuilder in

let _ = Llvm.build_ret sum_param ibuilderin

…
```

See file local_variables.ml (gen_foo function)

We allocate the space of the stack (`param_stack`) and use store and load to initialize the memory and to load its value on a register

Example of local variables (2)

```
let local_var = Llvm.build_alloca i32t "local" ibuilder in
let global_a = Llvm.lookup_global "a" llmodule |> Option.get in
let load_a = Llvm.build_load global_a "load_a" ibuilder in
let sum_10 = Llvm.build_add load_a (Llvm.const_int i32t 10) "sum10" ibuilder in
let mul_2  = Llvm.build_mul sum_10 (Llvm.const_int i32t 2)  "mul2" ibuilder in
let _ = Llvm.build_store mul_2 local_var ibuilder  in
let _ = Llvm.build_ret_void ibuilder  in
...
```

See file local_variables.ml (gen_bar function)

We allocate the space of the stack (`local_var`) and `lookup_global` to take a reference to the global variable `a`

Getting the address of an element

There are functions that allows building variants of `getelementptr` instruction

- `build_gep p indices name b` creates a `%name = getelementptr %p, indices...` instruction at the position specified by the instruction builder `b`
- `build_in_bounds_gep p indices name b` creates a `%name = gelementptr inbounds %p, indices...` instruction at the position specified by the instruction builder `b`
- `build_struct_gep p idx name b` creates a `%name = getelementptr %p, 0, idx` instruction at the position specified by the instruction builder `b`

See previous lecture on the use of `getelementptr`

Example of accessing array (1)

```
let index = Llvm.param fundef 0  in

let address = Llvm.build_gep global_a [|(Llvm.const_int i32t 0) ; index |] "access" ibuilder in

let _ = Llvm.build_store (Llvm.const_int i32t 2) address ibuilder in

let load_e = Llvm.build_load address "load" ibuilder in

let _ = Llvm.build_ret load_e  ibuilder  in

…
```

- See the file array_access.ml (gen_foo function)
- We assume to have a global array (`global_a`), we store 2 into the element `index` and return the value just stored

Example of accessing array (2)

```
let a2i32t = Llvm.array_type i32t 2 in

let array_local = Llvm.build_array_alloca a2i32t (Llvm.const_int i32t 2) "array_alloca" ibuilder in

let address1 = Llvm.build_in_bounds_gep array_local [|Llvm.const_int i32t 0) ; (Llvm.const_int i32t 1)|] "gep"
ibuilder in

let _ = Llvm.build_store (Llvm.const_int i32t 10) address1 ibuilder in

let load_e = Llvm.build_load address1 "load" ibuilder in

let _ = Llvm.build_ret load_e ibuilder in

...
```

- See the file array_access.ml (gen_bar function)
- We allocate an local array (`array_local`), we store 10 into the element `1` and return the value just stored

Calling functions

The following function allows us to invoke a function

> `build_call fn args name b` creates a `%name = call %fn(args...)` instruction at the position specified by the instruction builder `b`

The function `fn` is a llvalue representing a function either declaration (prototype) or definition

`declare_function name ty m` returns a new function of type `ty` and with name `name` in module `m`

Example of calling an external function (1)

Declaring puts

```
let i32t = Llvm.i32_type llcontext  in
let char_ptr = Llvm.i8_type llcontext |> Llvm.pointer_type  in
let puts_t = Llvm.function_type i32t [|char_ptr|]  in
Llvm.declare_function  "puts" puts_t llmodule
```

See file call_ext_fun.ml (in function declare_puts)

- `Llvm.pointer_type` given a type returns the corresponding pointer type
- `Llvm.declare_function` returns the llvalue correspoding to the prototype of puts

Example of calling an external function (2)

Calling puts

```
let str = Llvm.build_global_string "Hello World\n" "msg" ibuilder in
let zero = Llvm.const_int (Llvm.i64_type llcontext) 0 in
let pstr = Llvm.build_gep str [|zero; zero|] "gep" ibuilder in
let _ = Llvm.build_call puts [|pstr|] "call" ibuilder in
let _ = Llvm.build_ret_void ibuilder in
```

See file call_ext_fun.ml (in function define_main)

- `Llvm.build_global_string` builds a string literal (array of `i8`) in the global context
- `puts` represents the prototype of puts
- `puts` expects a `i8*`, so we use gep to get the pointer to the first element of the string

Conditions

Recall that there are two conditions instructions `icmp` and `fcmp` that can be built by

- `build_icmp pred x y name b` creates a `%name = icmp %pred %x, %y` instruction at the position specified by the instruction builder `b`, where `pred` is a value of type `Icmp.t`, e.g., `Eq`, `Ne`, `Ugt`, `Uge`, etc. (see documentation for others)
- `build_fcmp pred x y name b` creates a `%name = fcmp %pred %x, %y` instruction at the position specified by the instruction builder `b`, where `pred` is a value of type `Fcmp.t`, e.g., `Oeq`, `One`, `Ugt`, `Uge`, etc. (see documentation for others)

Example of comparison

```
let param0 = Llvm.param defeq 0 in
let param1 = Llvm.param defeq 1 in
let icmp = Llvm.build_icmp Llvm.Icmp.Eq param0 param1  "icmp" ibuilder  in
let _ = Llvm.build_ret icmp ibuilder  in
```

…

See file fun_equals.ml (in function define_equals)

We return the result of the comparison

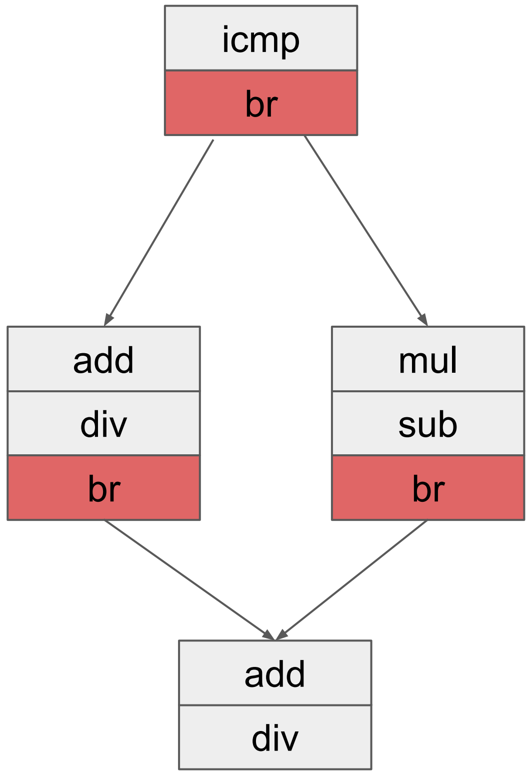
Conditional and unconditional jumps

Jump instruction can be built by

- `build_br bb b` creates a `br %bb` instruction at the position specified by the instruction builder `b`, where `bb` represents the basic block where we want to jump to
- `build_cond_br cond tbb fbb b` creates a `br %cond, %tbb, %fbb` instruction at the position specified by the instruction builder `b`, where `tbb` represents the basic block where we jump when the condition is true, `fbb` when it is false

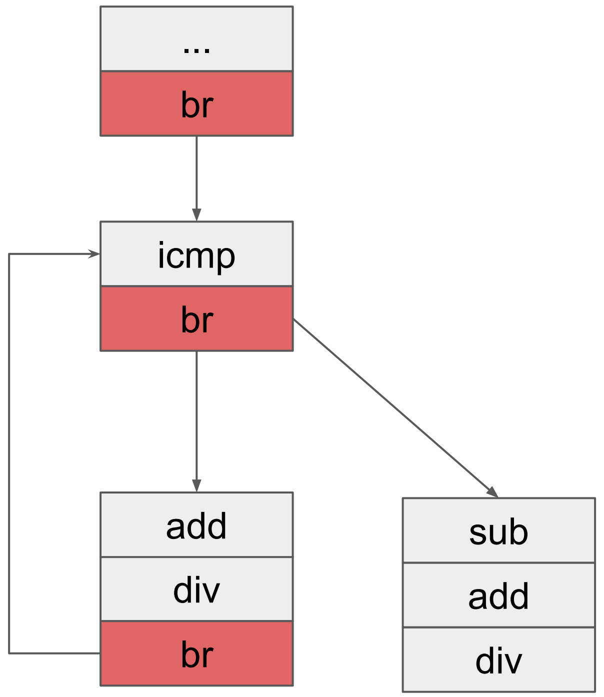
Compile an if-then-else statement

- Call the current basic block CB
- Generate basic blocks for the then branch TB, for the else branch EB and for when the control-flow join FB
- Fill TB and EB with instructions
- Generate as last instruction in TB and EB an unconditional jump to TF
- Generate as last instructions the comparison and the conditional branch in CB



Compile a loop

- Call the current basic block B
- Generate basic blocks for the condition CB, for the loop body BB and for continuation of the program FB
- Generate a unconditional branch in B to CB
- Fill CB with an comparison instruction and with a conditional branch to BB and FB
- Fill BB with instructions and generate a unconditional branch to CB as last instruction



Example of conditional (1)

```
let bthen = Llvm.append_block llcontext  "then" defabs in
let belse = Llvm.append_block llcontext  "else" defabs in
let bcont = Llvm.append_block llcontext  "cont" defabs in
let load_param = Llvm.build_load alloc_param  "load" ibuilder in
let zero = Llvm.const_int i32t  0 in
let icmp = Llvm.build_icmp Llvm.Icmp.Ult load_param zero  "icmp" ibuilder in
let _ = Llvm.build_cond_br icmp bthen belse ibuilder  in
...
```

See fun_abs.ml (in function define_abs)

- We create new basic blocks with `Llvm.append_block`
- The conditional jump uses the result of the comparison and target the blocks for then and else branches

Example of conditional (2)

```
Llvm.position_at_end bthen ibuilder;
let neg = Llvm.build_sub zero load_param  "neg" ibuilder in
let _ = Llvm.build_store neg alloc_param ibuilder  in
let _ = Llvm.build_br bcont ibuilder  in
Llvm.position_at_end belse ibuilder;
let _ = Llvm.build_br bcont ibuilder  in
Llvm.position_at_end bcont ibuilder;
let load = Llvm.build_load alloc_param  "load" ibuilder in
let _ = Llvm.build_ret load ibuilder  in
…
```

See fun_abs.ml (in function define_abs)

We use `Llvm.position_at_end` to move the instruction builder at the end of the basic block we want to work on

Phi instructions

The following functions allow us to create and manipulate the phi instructions

- `build_phi incoming name b` creates a `%name = phi %incoming` instruction at the position specified by the instruction builder `b` and where `incoming` is a list of `(llvalue, llbasicblock)` tuples
- `build_empty_phi ty name b` creates a `%name = phi %ty` instruction at the position specified by the instruction builder `b`
- `add_incoming (v, bb) pn` adds the value `v` to the phi node `pn` for use with branches from `bb`

Example of phi instruction

```
let icmp = Llvm.build_icmp Llvm.Icmp.Ult param zero "icmp" ibuilder in
let _ = Llvm.build_cond_br icmp bthen belse ibuilder  in
Llvm.position_at_end bthen ibuilder;
let neg = Llvm.build_sub zero param  "neg" ibuilder in
let _ = Llvm.build_br bcont ibuilder  in
Llvm.position_at_end belse ibuilder;
let _ = Llvm.build_br bcont ibuilder  in
Llvm.position_at_end bcont ibuilder;
let load = Llvm.build_phi [(neg, bthen) ; (param, belse)]  "phi" ibuilder in
let _ = Llvm.build_ret load ibuilder  in
…
```

See file fun_abs_phi.ml (in function define_abs)

The phi instruction merge the flow and data from `bthen` and `belse`

Other function in API

The OCaml API covers many LLVM classes

- Functions to manipulate attribute, e.g., volatile, linkage
- Code transformations
- Compiler passes
- Module for specific target architecture
- Functions to invoke the execution engine, aka JIT compiler

See the documentation

Example: First Order FUN language

We consider a restricted version of FUN

Major changes:

- A program is made by a (possible empty) sequence of function definitions and by an main expression
- No high-order functions
- Function definitions are not expressions
- Functions cannot be nested
- In function application the first component is the name of a function:

ID Expr

FUN: code generation strategy

- For each function definition in the source language we generate a corresponding definition in the LLVM IR
- We generate the definition of a `void main()` function to contain the code generated for the main expression
- FUN is a pure functional language with simple data types, thus, we do not need to allocate memory on the stack: all data are kept in registers
- During the code generation we maintain an environment mapping identifiers to llvalues that may represent function definitions, formal parameters and local identifiers

FUN: code generation

See the file fun_codegen.ml

Mainly three functions handle code generation:

- `codegen_main llmodule env e` generates the definition of a main function printing the result of the evaluation of `e`.
- `codegen_fundecl llmodule env fundecl` generates the definition of a main function fundecl
- `codegen_expr current_fun env ibuilder e` generates the code for the expression `e` where `current_fun` is the current function and `ibuilder` is the builder to use for generating LLVM IR instructions

Conclusion

- LLVM API in OCaml
  - Module declaration
  - Function declaration
  - Statement and expressions
- Code generation for first order FUN

References

- OCaml bindings documentation
- Kaleidoscope: Implementing a Language with LLVM in Objective Caml