Compilation 2024

Dolphin: phase 5

Amin Timany
timany@cs.au.dk

LLVM -- for translating aggregates

- Structure types
- Fixed-size arrays
- Named types
- Global Variables
- String variables
- Casting
- Pointer to integer conversion
- Computing physical size of types
- getelementptr (Gep)

Translating Types

- Start translation by translating types to LLVM --

```
let trans_type tp =
  match tp with
  | TAst.Determined (TAst.Void _) -> Ll.Void
  | TAst.Determined (TAst.Int _) -> Ll.I64
  | TAst.Determined (TAst.Bool _) -> Ll.I1
  | TAst.Determined (TAst.Byte _) -> Ll.I8
   ...
```

Flexible Array Elements

- In C (as of C99) we can pack an array together with its length into a single struct by having an array field of **unspecified length** as the **last** field.

```
struct array { int64_t len; char contents[]; };
```

- On the LLVM side, this is represented as:

```
%array_type = type {i64, [0 x i8]}
```

- We use pointers to these types to represent both Dolphin's arrays and strings

Translating Arrays & Strings

- Start translation by translating types to LLVM --

```
let trans_type tp =
  match tp with
  | TAst.Determined (TAst.Void _) -> Ll.Void
  | TAst.Determined (TAst.Int _) -> Ll.I64
  | TAst.Determined (TAst.Bool _) -> Ll.I1
  | TAst.Determined (TAst.Byte _) -> Ll.I8
  | TAst.Determined (TAst.Str _) ->
      Ll.Ptr (Ll.Namedt (Sym.symbol "array_type"))
  | TAst.Determined (TAst.Array _) ->
      Ll.Ptr (Ll.Namedt (Sym.symbol "array_type"))
    ...
```

Translating String Literals

```
int main () {
    var x = "Hello World\n";
    output_string(x, get_stdout());
    return 0;
}
```

The Dolphin program above is translated to the following LLVM -- program

```
%dolphin_record_stream = type {  }
%array_type = type { i64, [0 x i8] }

@string_literal$1 = global { i64, [12 x i8] } {i64 12, [12 x i8] c"Hello World\0A"}

declare void @output_string(%array_type*, %dolphin_record_stream*)
declare %dolphin_record_stream* @get_stdout()

define i64 @dolphin_fun_main () {
 %x$0 = alloca %array_type*
 %conv_string_literal_packed$2 = bitcast { i64, [12 x i8] }* @string_literal$1 to %array_type*
 store %array_type* %conv_string_literal_packed$2, %array_type** %x$0
 %load_local_var$3 = load %array_type*, %array_type** %x$0
 %call$4 = call %dolphin_record_stream* @get_stdout ()
 call void @output_string (%array_type* %load_local_var$3, %dolphin_record_stream* %call$4)
 ret i64 0
after_return$5:
 unreachable
}
```

Translating Array Creation

- Use the runtime function

```
struct array *allocate_array(int32_t elem_size, int64_t numelems, void* contents)
```

- Compute the size of elements in LLVM
- Determine the default value based on the type
- Allocate space for the default value using LLVM's `alloca`
- Store the default value into the allocated space
- Pass the pointer to the default value along with size of array's elements and its length to the `allocate_array`
- Note how `allocate_array` returns the correct type!

Translating `length_of`

- `length_of` acts on arrays and strings
- These are both represented as pointers to `%array_type*` in LLVM
- Use Gep to read the length field of `%array_type`

Translating the Lval `a[i]`

- Translate `a`, it should produce an LLVM operand of type `%array_type*`
- Use Gep to get a pointer to the filed `contents`
  - Treat this as a point type `i8*`
  - Bitcast it to a pointer of type `t*` where `t` is the translation of the Dolphin type `A` such that array `a` has type `[A]` in Dolphin (use `trans_type`)
- Use Gep to index into the obtained pointer of type `t*`

- **NOTE**: The `trans_lval` function should **return a pointer**; the `trans_expr` function should it appropriately

Translating nil

- Translate nil to LLVM’s null
- We update trans_type to support TAst.Undetermined _

```
let trans_type tp =
  match tp with
  | TAst.Determined (TAst.Void _) -> Ll.Void
  | TAst.Determined (TAst.Int _) -> Ll.I64
  | TAst.Determined (TAst.Bool _) -> Ll.I1
  | TAst.Determined (TAst.Byte _) -> Ll.I8
  | TAst.Determined (TAst.Str _) ->
      Ll.Ptr (Ll.Namedt (Sym.symbol "array_type"))
  | TAst.Determined (TAst.Array _) ->
      Ll.Ptr (Ll.Namedt (Sym.symbol “array_type”))
  | TAst.Undetermined _ -> Ll.Ptr Ll.I8
   ...
```

Translating Record Declarations

- We translate record declarations into a **named struct types** in LLVM --

- We translate lists

```
record list {head : int; tail : list;}
```

into

```
%dolphin_record_list = type { i64, %dolphin_record_list* }
```

- Use for `trans_type` translating types of fields

Translating Types Including Records

Sometimes we need the type of records not wrapped in Ll.Ptr

```
let trans_type raw_records tp =
  match tp with
  | TAst.Determined (TAst.Void _) -> Ll.Void
  | TAst.Determined (TAst.Int _) -> Ll.I64
  | TAst.Determined (TAst.Bool _) -> Ll.I1
  | TAst.Determined (TAst.Byte _) -> Ll.I8
  | TAst.Determined (TAst.Str _) ->
     Ll.Ptr (Ll.Namedt (Sym.symbol "array_type"))
  | TAst.Determined (TAst.Array _) ->
     Ll.Ptr (Ll.Namedt (Sym.symbol “array_type”))
  | TAst.Undetermined _ -> Ll.Ptr Ll.I8
  | TAst.Determined (TAst.Record {recordname = RecordName {sym; _}}) ->
    let raw_res = Ll.Namedt (encode_record_name sym) in
    if raw_records then raw_res else Ll.Ptr raw_res
  | TAst.Determined TAst.ErrorType -> assert false
```

Translating Field Access Lvals

- To translate `a.f`, first translate `a`
- We know the Dolphin type `A` of `a` from semantic analysis
- Translate `A` into an LLVM type (raw record type) `t`
- Figure out the index of field `f` in the LLVM type `t`
- Use Gep to obtain a pointer to field `f`

Translating Record Creation

- Use the runtime function

```
void *allocate_record(int32_t size)
```

- Compute the size of the record type in LLVM
- Call the `allocate_record` function to allocate space for the array
- The returned pointer has type `i8*` on the LLVM side
- Bitcast the obtained pointer into a pointer of the record
- For each field
  - Translate the initialization expression for that field
  - Compute the address of the field using Gep (as on the previous slide)
  - Write the computed initialization value into the field

Translating comparison

- Check the underlying type
    - if it is two strings being compared, use the runtime function for comparing strings
    - If it is not strings
        - If the comparison is <, <=, <, or >=
            - The objects being compared are integers; use LLVM's `icmp` instruction
        - Otherwise, the comparison is == or `!=`
            - Use LLVM's `icmp` instruction
            - The type of objects being compared (needed by `icmp`) can be obtained by calling `trans_type`

Stdlib Types and their Translation

- The standard library features types that should be represented in Dolphin programs, e.g., the `stream` type
- These types are represented (hardcoded) as **empty records** in Dolphin, and translated into LLVM as such
  ```
  %dolphin_record_stream = type {}
  ```
- **Note**: Semantic analysis should prohibit creating instance of these records, i.e., the following should be rejected with an error:
  ```
  var z = new stream {};
  ```