Compiler report - Project COCass

Villani Neven, ENS Paris-Saclay, Programmation 1

January 6, 2021

1 Major design choices

1.1 Compiler passes

The compiler does the following passes:

1. lex and parse source code into ASM
2. reduce expressions
3. generate abstract assembler
4. translate to text
5. print aligned instructions
6. assemble and link

Reduction and generation are partly merged : each time an expression is encountered it is reduced (only once) before being generated.
The parsing step produces a list of declarations, which the reduction slightly modifies. The generation outputs the abstract assembler defined in `generate.ml`. Translation and printing combined produce ascii text. The assembly step is external.

1.2 File distribution

- `cAST.ml`: text to AST translation
- `compile.ml`: AST to `program`
- `generate.ml`: `program` to `alignment` to string representation
- `reduce.ml`: expression reduction
- `pigment.ml`: color management

1.3 Scope management

The scope is managed throughout `compile.ml` with the three parameters

- `envt`, an associative list that yields a location for each identifier
- `depth`, an integer that tells the current number of local or temporary variables on the stack
- `va_depth`, either `None` or the number of non-optional parameters

To locate a variable, `envt` is scanned and yields a `location`:

```
type location =
    | Stack of int (* variable is on the stack at the given depth *)
    | Const of int (* known constant value *)
    | Globl of string (* global variable *)
    | FnPtr of string (* function pointer *)
    | Hexdc of string (* hexadecimal constant *)
    | Regst of register (* variable is in a register *)
    | Deref of register (* variable is at the address given by a register *)
    | Index of register * register (* variable is at an index with offset given by the two registers *)
```

1.4 Register distribution

Throughout the main function of `compile.ml`, the following conventions are used:

- `RAX` contains the last evaluated expression
- `RDI` contains the last evaluated address
- `RCX` is an extra register for 2-register operations
- `R10` is used whenever a function pointer is needed

1.5 Tag management

Tags also have rules:

- `.LC`n are strings
- `.EX`n are exception names (except for $n \in \{0, 1\}$ reserved for exception formatting strings)
- `.eaddr` is the address of the current exception handler
- `.ebase` is the base pointer of that handler
- f`.`i`_`l is the usual format for the i'th tag in function f of type l. Values for l include `return`, `loop_start`, `switch_done`, ...

Some parameters are passed around in the generating functions to record which tags to jump to at various steps.

1.6 Function skeleton

All functions have the same basic structure, they :

- start by storing their callee-saved registers
- keep their frame pointer equal to their base pointer except during subroutine calls
- have a `return` tag that they jump to in order to return

1.7 For convenience

To avoid some mistakes and avoid redefining all common constant values, the compiler includes a fixed list of known functions selected from the standard library and some constant values. These functions are spell-checked, and their number of arguments is validated.
Constants include common values such as `NULL` and `EOF` as well as non-standard ones like `BYTE` (255).

2 Issues encountered

During the development process very few issues were serious enough to slow down notably the progress of the compiler. Some minor inconveniences were:

- figuring out the right assembler instructions for lack of beginner-friendly documentation on the internet
- experimenting with the stack to figure out the exact location of all arguments
- setting up the assembler header instructions (`.text`, `.align 8`, ...)

More anecdotally:

- ~30min spent figuring out why `printf` segfaults when `rax` is not zeroed before the call
- ~1h spent identifying and fixing a stack alignment issue (`rsp` has to be a multiple of 16)

3 Additions

The following constructs were added to the supported language:

3.1 Unary operators

`M_DEREF`: accesses the address given by the expression.
Examples:

- `*x` → `OP1(M_DEREF, VAR "x")`
- `*(x+1)` → `OP1(M_DEREF, OP2(S_ADD, VAR "x", CST 1))`

`M_ADDR`: yields the address of an expression
Examples

- `&x` → `OP1(M_ADDR, VAR "x")`
- `&x[10]` → `OP1(M_ADDR, OP2(M_INDEX, VAR "x", CST 10))`
- `&*x` → `OP1(M_ADDR, OP1(M_DEREF, VAR "x"))`

Errors

- `&10`, `&(x+1)`, `&"abc"` → 'Indirection needs an lvalue'

Note:

- ∀`e`, `OP1(M_ADDR, OP1(M_DEREF, e))` is equivalent to `e`
- ∀`e` with an address, `OP1(M_DEREF, OP1(M_ADDR, e))` is equivalent to `e`

3.2 Binary operators

The following binary operators were added: `S_SHL`, `S_SHR`, `S_OR`, `S_XOR`, `S_AND` Examples

- `x << 2` → `OP2(S_SHL, VAR "x", CST 2)`
- `x >> 2` → `OP2(S_SHR, VAR "x", CST 2)`
- `x | 2` → `OP2(S_OR, VAR "x", CST 2)`
- `x ^ 2` → `OP2(S_XOR, VAR "x", CST 2)`
- `x & 2` → `OP2(S_AND, VAR "x", CST 2)`

3.3 Comparisons

To simplify some expressions, the reduction step is allowed to perform the following transformations.
For any `a`, `b`:

- `EIF(CMP(C_EQ, a, b), 0, 1)` → `CMP(C_NE, a, b)`
- `EIF(CMP(C_LE, a, b), 0, 1)` → `CMP(C_GT, a, b)`
- `EIF(CMP(C_LT, a, b), 0, 1)` → `CMP(C_GE, a, b)`

3.4 Extended assignment

Examples

- `x += 2` → `OPSET(M_ADD, VAR "x", CST 2)`
- `x *= 2` → `OPSET(M_MUL, VAR "x", CST 2)`
- etc...

Errors

- `x []= 2` → parsing error
- `2 += 2` → 'Extended assignment needs an lvalue'

3.5 Control flow

`CBREAK` (exit loop or switch), `CCONTINUE` (skip to next iteration of loop), `CTHROW` (raise exception)
Examples

- `break;` → `CBREAK`
- `continue;` → `CCONTINUE`
- `throw E(x);` → `CTHROW("E", VAR "x")`
- `throw E;` → `CTHROW("E", VAR "NULL")`

Errors

- `try { break; }` → 'break may not reach outside of try'
- `try { continue; }` → 'continue may not reach outside of try'

3.6 Declarations

`CLOCAL` is used to declare a local variable in the middle of a block. Examples

- `int x;` → `CLOCAL[("x", None)]`
- `int x, y;` → `CLOCAL[("x", None), ("y", None)]`
- `int x = 1;` → `CLOCAL[("x", Some (CST 1))]`

3.7 Switch

`CSWITCH` declares a switch block. Example:

```
switch (x) {
    case 1: 1;
    case 2: 2;
    default: 3;
}
```

parses to

```
CSWITCH (VAR x, [
    (1, CBLOCK [CEXPR (CST 1)]);
    (2, CBLOCK [CEXPR (CST 2)])
], CBLOCK [CEXPR (CST 3)])
```

3.8 Try

`CTRY` declares an exception handling block. Example:

```
try { 1; }
catch (E x) { 2; }
catch (F _) { 3; }
catch (G) { 4; }
finally { 5; }
```

parses to

```
CTRY (CBLOCK [CEXPR (CST 1)], [
    ("E", "x", CBLOCK [CEXPR (CST 2)]);
    ("F", "_", CBLOCK [CEXPR (CST 3)]);
    ("G", "_", CBLOCK [CEXPR (CST 4)])
], CBLOCK [CEXPR (CST 5)])
```