Compiler report - Project COCass

VILLANI Neven, ENS Paris-Saclay, PROGRAMMATION 1

January 8, 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

Roles are distributed as such:

- `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`, which denotes any of the supported ways to access the value/address of a variable.

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).

1.8 Exception management

Exceptions are handled roughly as follows.
When a `try` block is encountered a handler is built: the current value for `%rbp` is stored in `.ebase`, the tag to jump to in the case of a `throw` is stored in `.eaddr`. A similar handler is set up at the start of main.
If the block exits without an exception, the handler is restored to its previous value (`.eaddr` and `.ebase` were saved on the stack during setup) and the `finally` clause is executed.
If an exception is raised the `throw` instruction restores the base pointer and jumps to the handler address. The handler is saved in temporary registers then restored to its previous value. Handler name (determined by the address at which it is stored) is compared sequentially to the handlers. A variable binding is created with the value thrown. There are two exceptions to these rules: `_` as an exception name can match any exception (but can't be thrown); `_` as a variable produces no binding. A `catch (_) {}` statement is a surefire way to ensure the program will not crash.
The emergency handler (setup during main) flushes the output then prints its own error message with the name and value of the exception. It uses a criteria on the value to guess whether it should be printed as an integer or a string. Strings created dynamically will wrongly be displayed as integers, and integers that happen to equal an address at which a static string is stored will wrongly be displayed as strings.

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 x86 instructions for lack of beginner-friendly documentation on the internet (at least 1h)
- experimenting with the stack to figure out the exact location of all arguments (several hours)
- setting up the assembler header instructions (`.text`, `.align 8`, ...) (at least 30min)

More anecdotally:

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

Although I believe I succeeded in specifying the complete (apart from variadic arguments) semantics of extended C−−, I tried and failed to implement them in Coq. The main reason for that failure is the semantic abstraction of the memory: each value being spread over 8 bytes makes it very difficult to establish the required properties of the memory.

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)])
```

3.9 Other

Some other features:

- Variadic arguments are supported. They are declared with ... then can be used with `va_start` and `va_arg`.
- Function pointers are supported. The original goal was to enable calling `qsort`.
- `switch` uses a jump table in the form of a binary tree. Its number of comparisons is logarithmic with regards to the number of `case` statements.
- `void` and `long` have been added as valid type declarations. They are equivalent to `int` but allow being more explicit.
- The `assert` builtin has been implemented. It may raise an `AssertionFailure` exception which the user can catch.
- Color output is available by default, `--no-color` disables it.
- Expressions are reduced when possible by default, `--no-reduce` disables it.

4 Modifications

The following changes were made to existing constructs of the AST.
Each comes with an explanation of why it was deemed necessary in the context of other additions to the language.

4.1 Assignments

`SET` replaces both `SET_VAR` and `SET_ARRAY` as well as what was for some time `SET_DEREF`.

- `x = 1;` $\not\to$ `SET_VAR("x", CST 1)` $\to$ `SET(VAR "x", CST 1)`
- `t[0] = 1;` $\not\to$ `SET_ARRAY("t", CST 0, CST 1)` $\to$ `SET(OP2(S_INDEX, VAR "t", CST 0), CST 1)`
- `*x = 1` $\not\to$ `SET_DEREF("x", CST 1)` $\to$ `SET(OP1(M_DEREF, VAR "x"), CST 1)`

Justification:
With the addition of `*x`, `SET_DEREF(x, e)` was first added but required much code duplication (all code related to assignment needed to appear thrice).
At first, `t[x][y] = 1;` was a parsing error, even though `t[x][y]` was a valid expression.
Since `M_DEREF` added the horrible workaround `*(&t[x][y]) = 1;`, allowing any expression to be assigned to, I decided it was time to allow more expressions to be treated as lvalues. Changing assignment was deemed the best course of action.
At the same time, former constructors `OPSET_VAR`, `OPSET_ARRAY` and `OPSET_DEREF` were all merged into `OPSET`.

4.2 Blocks

This code

```
{
    int x, y;
    x = 1;
}
```

used to parse to

```
CBLOCK (
    [CDECL "x", CDECL "y"],
    [CEXPR (SET_VAR ("x", CST 1))]
)
```

It now yields

```
CBLOCK [
    CLOCAL [("x", None); ("y", None)];
    CEXPR (SET (VAR "x", CST 1))
]
```

Justification:
As soon as `int x;` and `int x = 1;` were allowed anywhere in the code and not just at the start of blocks, it made no more sense to have blocks carry information on the variables defined inside of them in a different form.

4.3 Loops

- `for (e1; e2; e3) { c }` $\not\rightarrow$ `e1; CWHILE (e2, c @ [e3])` $\rightarrow$ `e1; CWHILE (e2, c, e3, true)`
- `while (e) { c }` $\not\rightarrow$ `CWHILE (e, c)` $\rightarrow$ `CWHILE (e, c, ESEQ [], true)`
- `do { c } while (e);` $\not\rightarrow$ (parsing error) $\rightarrow$ `CWHILE (e, c, ESEQ [], false)`

Justification:
The addition of do-while required some information on whether the test should be done at the start of the loop, a boolean was chosen.
At the same time, wanting to implement `break` and `continue`, I deemed it necessary to separate the body block and the finally clause of `for`. This is what the third argument accomplishes.

4.4 Declarations

New declaration typing:

```
type top_declaration =
  | CDECL of var_declaration * loc_expr option
  | CFUN of var_declaration * var_declaration list * loc_code
and var_declaration = Error.locator * string
and local_declaration = var_declaration * loc_expr option
```

Justification:
The addition of optional initialisation values made no sense for function parameters, also `CDECL | CFUN` matches were required in many places where `CFUN` were impossible anyway. This resulted in too many ignored parameters or matches of the form `| _ -> failwith "unreachable"` for my taste.
To streamline the typing `var_declaration` was renamed to `top_declaration` as an indication that it denotes a toplevel declaration. The name `var_declaration` was reserved for function arguments, and `local_declaration` for local variables.

```
int x;              // CDECL ((_,"x"), None)
int y = 1;          // CDECL ((_,"y"), Some (CST 1))

int foo (int z) {   // CFUN (_, "foo", [(_,"z")], ...)
    int i;          // CLOCAL [((_,"i"), None)]
    int j = 0;      // CLOCAL [((_,"j"), Some (CST 0))]
}
```

5 Conclusion

I very much enjoyed working on this compiler, although I had a bit of a hard time at first trying to adapt to this assembler different from the one I knew previously.
I do find it a bit disappointing that `gcc` is a dependency, even if only for the assembly and linking steps, if I had a bit more time I probably would attempt to make `mcc` more autonomous.

Development process was somewhat test-driven: the typical workflow consisted of deciding on a syntax feature to support, writing several C files using this feature, designing its AST representation, adding it to the grammar of the parser (with the help of `cprint` to ensure correct parsing), implementing its compilation file-by-file, running the automatic tester to detect and fix regressions.
The same automatic tester and the wide range (80+ files, 330+ tests) of tests were incredibly useful in checking that no issues were introduced during big refactorings.

The aspect of the compiler I am most content with is without a doubt the abstract assembler. Its interface allowed me to have a less error-prone way of generating assembler (in particular having type-checked assembler instructions), with a lot fewer `sprintf` polluting the compilation logic than otherwise. Without it I have no doubt that the alignment and syntax highlighting of the assembler dump would have been impossible.