Intermediate Representation

Enrico Deiana, Emanuele Del Sozzo

Introduction

We give here a brief introduction on how the *el* compiler is implemented so far what can be done to improve it.

Right now there isn't a proper Intermediate Representation (IR) code generation (it is like *PyPy* before *Just-in-Time* compiler was introduced). *el* code is tokenized by a lexer (*lexer.go*), parsed by Bison (*parser.y*), and an Abstract Syntax Tree (AST) of the code is built; then, evaluating each node, the code is compiled. Each statement and operation is executed in *go* (e.g. an *el* sum becomes a sum in *go* and then the result is returned). In such way the Abstract Syntax Tree itself is used as an high level IR.

This approach has the advantage of simplicity (the AST itself is the IR), but using an explicit IR (which is the "standard way" to build a compiler) turns into a simpler and more readable representation of the code.

IR Instructions

All the instructions are defined by this data structure:

```
type Intruction struct{
        op      int
        arg1    Symbol
        arg2    Symbol
        result  Symbol
        true    *Instruction
        false   *Instruction
        next    *Instruction
}
```

Binary Operations

The binary operations defined in **EL** are:

- arithmetical operations (*SUM*, *SUB*, *MUL*, *DIV*);
- logic operations (*AND*, *OR*);
- comparison operations (*EQUAL*, *NOT_EQUAL*, *LOWER*, *GREATER*, *LOWER_EQUAL*, *GREATER_EQUAL*);
- access operations (*VALUE_ACCESS*, *SQUARE_ACCESS*).

An example of arithmetical operation is:

EL instruction:

$x=$y+$w+$z

Intermediate representation:

$Instr_1$ op: ADD arg1: \$y arg2: \$w result: \$$x_1$ true: NULL false: NULL next: $Istr_2$
$Instr_2$ op: ADD arg1: \$$x_1$ arg2: \$z result: \$$x_2$ true: NULL false: NULL next: $Istr_3$
$Instr_3$ op: ASSIGNMENT arg1: \$$x_2$ arg2: NULL result: \$x true: NULL false: NULL next: $Istr_4$
$Instr_4$ ...

An example of access operation is:
EL instruction:

$x=$a[$i]

Intermediate representation:

$Instr_1$ op: SQUARE_ACCESS arg1: \$a arg2: \$i result: \$$x_1$ true: NULL false: NULL next: $Istr_2$
$Instr_2$ op: ASSIGNMENT arg1: \$$x_1$ arg2: NULL result: \$x true: NULL false: NULL next: $Istr_3$
$Instr_3$ ...

Unary Operations

The unary operations defined in **EL** are:

- arithmetic operation (UNARY_MINUS);
- logic operation (NOT);
- operations on addresses (ASSIGNMENT).

An example is:
EL instruction:

$x=-$y

Intermediate representation:

$Instr_1$ op: UNARY_MINUS arg1: \$y arg2: NULL result: \$$x_1$ true: NULL false: NULL next: $Istr_2$
$Instr_2$ op: ASSIGNMENT arg1: \$$x_1$ arg2: NULL result: \$$x$ true: NULL false: NULL next: $Istr_3$
$Instr_3$ ...

Unconditional Jumps

An unconditional jump occurs when there is a *jump* that does not depend on the evaluation of any condition. Keywords such as *break* and *continue* are examples of unconditional jumps. Here's an example of *break* keyword:
EL instruction:

```
for $i = 0; $i < 10; $i = $i + 1 {
  break
}
```

Intermediate representation:

$Instr_1$	op: ASSIGNMENT arg1: 0 arg2: NULL result: $\$i$ true: NULL false: NULL next: $Istr_2$
Label	*CONDITION*
$Instr_2$	op: LOWER arg1: $i arg2: 10 result: t_1 true: NULL false: NULL next: $Istr_3$
$Instr_3$	op: JUMP arg1: t_1 arg2: NULL true: *BODY* false: *OUT* next: NULL
Label	*BODY*
$Instr_4$	op: U_JUMP arg1: NULL arg2: NULL true: NULL false: NULL next: *OUT*
$Instr_5$	op: ADD arg1: $i arg2: 1 result: $\$i_1$ true: NULL false: NULL next: $Instr_6$
$Instr_6$	op: ASSIGNMENT arg1: $\$i_1$ arg2: NULL result: $\$i$ true: NULL false: NULL next: $Instr_7$
$Instr_7$	op: U_JUMP arg1: NULL arg2: NULL true: NULL false: NULL *CONDITION*
Label	*OUT*
$Instr_8$	...

Here's an example of *continue* keyword:
EL instruction:

```
for $i = 0; $i < 10; $i = $i + 1 {
  continue
}
```

Intermediate representation:

$Instr_1$	op: ASSIGNMENT arg1: 0 arg2: NULL result: $\$i$ true: NULL false: NULL next: $Istr_2$
Label	*CONDITION*
$Instr_2$	op: LOWER arg1: $i arg2: 10 result: t_1 true: NULL false: NULL next: $Istr_3$
$Instr_3$	op: JUMP arg1: t_1 arg2: NULL true: *BODY* false: *OUT* next: NULL
Label	*BODY*
$Instr_4$	op: U_JUMP arg1: NULL arg2: NULL true: NULL false: NULL next: *CONDITION*
$Instr_5$	op: ADD arg1: $i arg2: 1 result: $\$i_1$ true: NULL false: NULL next: $Instr_6$
$Instr_6$	op: ASSIGNMENT arg1: $\$i_1$ arg2: NULL result: $\$i$ true: NULL false: NULL next: $Instr_7$
$Instr_7$	op: U_JUMP arg1: NULL arg2: NULL true: NULL false: NULL next: *CONDITION*
Label	*OUT*
$Instr_8$	...

Conditional Jumps

A conditional jump occurs when there is a *jump* that depends on the evaluation of a condition. Control statements like *IF_THEN*, *IF_THEN_ELSE* and *FOR* use conditional jumps. Here's an example of *IF_THEN_ELSE* control statement:
EL code:

```
if $a < $b {
  $x=$y
}
else {
  $x=$z
}
```

Intermediate representation:

$Instr_1$	op: L_COMPARISON arg1: \$a arg2: \$b result: t_1 true: NULL false: NULL next: $Istr_2$
$Instr_2$	op: BNEQ arg1: t_1 arg2: NULL result: NULL true: *TRUE* false: *FALSE* next: NULL
Label	*TRUE*
$Instr_3$	op: ASSIGNMENT arg1: \$y result: \$x true: NULL *BODY* false: NULL next: *OUT*
Label	*FALSE*
$Instr_4$	op: ASSIGNMENT arg1: \$z result: \$x true: NULL *BODY* false: NULL next: *OUT*
Label	*OUT*
$Instr_5$	...

Function Call

The IR for a function call is as follows:

$x = some_function("%d", $k+1)

becomes:

$Instr_1$	op: PARAM arg1: "%d" arg2: NULL result: NULL true: NULL false: NULL next: $Istr_2$
$Instr_2$	op: ADD arg1: \$k arg2: 1 result: t_1 true: *TRUE* false: *FALSE* next: $Instr_3$
$Instr_3$	op: PARAM arg1: t_1 arg2: NULL result: NULL true: NULL false: NULL next: $Instr_4$
$Instr_4$	op: CALL arg1: some_function arg2: 2 result: t_2 true: NULL false: NULL next: $Instr_5$
$Instr_5$	op: ASSIGNMENT arg1: t_2 arg2: NULL result: \$x true: NULL false: NULL next: $Instr_6$
$Instr_6$	...

So, first of all the function parameters are evaluated and then the function call is performed.

Since we can have nested function calls, it is necessary to keep track of the number of parameters of each function; we do that using in the CALL instruction the number of needed parameters as second argument (the first one is the called function).

The run-time routines will handle procedure parameter passing, calls and return operations. The CALL instruction will execute the arg1 function using the arg2 needed parameters.