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*).

An example is:
EL instruction:

$$Istr_1 \quad \$x=\$y+\$w+\$z$$

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: ASSIGN arg1: \$$x_2$ result: \$x true: NULL false: NULL next: $Istr_4$
$Instr_4$	...

Unary Operations

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

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

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:

$Instr_1$	for \$i = 0; \$i < 10; \$i = \$i + 1 {
$Instr_2$	break }

Intermediate representation:

$Instr_1$	op: ASSIGN arg1: 0 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 true: *BODY* false: *OUT*
Label	*BODY*
$Instr_4$	op: U_JUMP next: *OUT*
$Instr_5$	op: ADD arg1: \$i arg2: 1 result: \$$i_1$ true: NULL false: NULL next: $Instr_6$
$Instr_6$	op: ASSIGN arg1: \$$i_1$ result: \$$i$ true: NULL false: NULL next: $Instr_7$
$Instr_7$	op: U_JUMP next: *CONDITION*
Label	*OUT*
$Instr_8$	...

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

$Instr_1$	for \$i = 0; \$i < 10; \$i = \$i + 1 {
$Instr_2$	continue }

Intermediate representation:

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