Intermediate Representation (IR)

Hodai Goldman, Lior Zur-Lotan and Avi Hayoun

December 24, 2015

Contents

IR is needed to achieve 2 goals:

1. Separating the front-end (source language) and the back-end (destination language) of the compiler to allow compiling multiple source languages to multiple target languages (modular compiler).

2. Optimizations independent of the source and destination languages (this covers most of the optimizations).

The IR we will use in this course is like a high level assembly language, with a subset of arithmetic and logical operations which instructions have at most 3 operands:

- Arithmetic operations: -, +, *, /
- Logical operations: =, (eq?), ≤

We will use this limited set to implement a larger set of operations implemented by any higher-level language.

1 Lowering examples

The process of Lowering is that of translating code from an sequence of ASTs of a higher-level language to an IR. We will use two functions to implement this process: CGen(expression) and Emit(IR-Code).

CGen is a recursive function which defines the proper translation of each type of expression, and Emit prints the output IR to a target file.

1.1 Simple Example

Source language:

1 \( x := 3; \)
2 \( y := 2 \times x + 7; \)
3 \( x := x \times 2 + y \times 2; \)
Using the CGen and Emit functions:

- **CGen**\( (x := 3;) \)
  - **Emit**\( (x := 3;) \)
- **CGen**\( (y := 2 \times x + 7;) \)
  - **Emit**\( (t0 := 2 \times x) \)
  - **Emit**\( (y := t0 + 7;) \)
- **CGen**\( (x := x \times 2 + y \times 2;) \)
  - **Emit**\( (t1 := x \times 2;) \)
  - **Emit**\( (t2 := y \times 2;) \)
  - **Emit**\( (x := t1 + t2;) \)

Which results in the output file:

```plaintext
x := 3;
t0 := 2 \times x;
y := t0 + 7;
t1 := x \times 2;
t2 := y \times 2;
x := t1 + t2;
```

**Remark.** \( t0, t1 \) and \( t2 \) are temporary intermediate values of computations.

### 1.2 Boolean example

Booleans are represented using numbers in IR: zero represents falseness, while every other value represents truthness. Note that this results in addition being equivalent to "or", multiplication being equivalent to "and", etc.

Lets start by lowering the \&\& (AND) operator:

```plaintext
x := (a \&\& b);
```

which, obviously, results in:

```plaintext
x := a \times b;
```

Now, let's implement the ! (Not) operator:
1.3 Conditions

Using the boolean operations, we can define conditionals and loops:

Source language:

```
if (x <= y) then
  z := x;
else
  z := y;

while (0 <= z) do
  z := z - 1;
```

IR:

```
t0 := x <= y;
ifZ t0 Goto _L0; /* if to evaluates to 0 == false */
z := x;
Goto _L1;
_L0:
z := y;
```
_L1:
_L2:
  t1 := 0 <= z;
  ifZ t1 Goto _L3;
  z := z - 1;
  Goto _L2;
_L3:

1.4 Function Definitions and Calls

Source:

```c
simpleFunc(x, y){
  return (x * y + 5) * 2;
}
main(){
  z := simpleFunc(17, 8);
}
```

IR:

```c
_SimpleFunc:
  t0 := p0 * p1;
  t1 := t0 + 5;
  t2 := t1 * 2;
  return t2;
_main:
  z := Call _SimpleFunc(17, 8);
```

Remark. \( p_0, p_1, \ldots \) are the indexed function parameters. So \( x \rightarrow p_0 \) and \( y \rightarrow p_1 \).

1.5 Putting it all together

Let's see how to handle a trivialized version of the exception mechanism:

Source:

```c
Div(x, y){
  if (y = 0) then
    Throw someException;
  else
    return x / y;
```
```java
main()
{
    try {
        Div(1, 0);
    } catch (Exception e) {
        exit(e);
    }
}

IR:

exceptionRef := 0; /* define a global variable and initialize it to 0 */
_Div:
t0 := p1 = 0;
IfZ t0 Goto _Throw;
t1 := p0 / p1;
return t1;
_Throw:
exceptionRef := Exception(0);
goto _Catch;

_main:
_try:
    Call Div(1, 0);
goto _Good;
_Catch:
    Call exit(exceptionRef);
_Good:
```

**Remark.** We need a way to let the exception result in a goto to the _Catch label. This can be achieved by passing the target label to the CGen() function when generating the IR for exceptions.

This code is intended for non-nested exceptions (which is almost never the case in real life). As you saw when learning about scoping, to implement real exception handling, an exception stack is needed.