Ficha prática 6

Compiladores

2018/2019

LLVM

1 Introdução

O projecto LLVM (Low-Level Virtual Machine) consiste numa colecção de ferramentas para compilação que inclui diversos elementos, nomeadamente um compilador de C/C++ (Clang), um optimizador e um gerador de código. É um projecto Open-Source recentemente impulsionado pela Apple como base para o XCode e para o desenvolvimento de apps para Mac e iOS.

Estas componentes são unidas por uma linguagem de representação intermédia de código (linguagem LLVM IR). Através desta linguagem intermédia, as ferramentas de optimização e geração de código para várias plataformas e arquiteturas podem ser reutilizadas por compiladores para várias linguagens (Figura 1). Esta ficha foca-se nessa linguagem (versão 3.8).

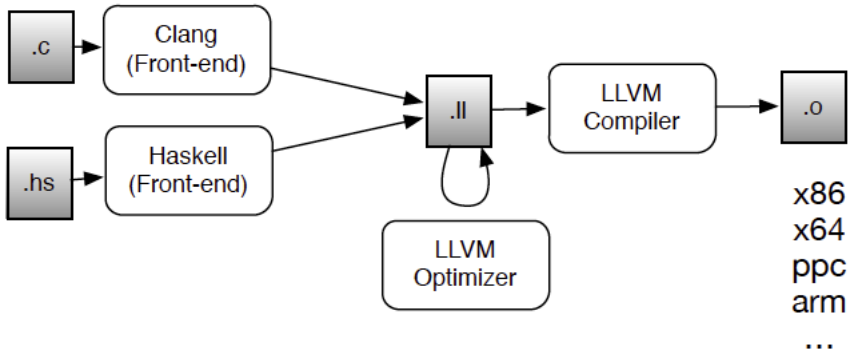

Figura 1: LLVM architecture

1.1 Executando o HelloWorld

Junto a este documento encontra-se um ficheiro chamado `hello.ll`. Esse ficheiro contem um exemplo de um Hello World escrito em LLVM IR. Para o executar diretamente, poderá usar o LLVM Interpreter (`lli`). Para tal, basta executar no terminal: `lli-3.8 hello.ll`. Deverá aparecer `Hello World` no stdout como resultado da execução do código LLVM.

Para compilar o código para um ficheiro objeto são necessários dois passos. O primeiro consiste em executar o LLVM Compiler (`llc`) usando o comando: `llc-3.8 hello.ll`. Este passo vai gerar um ficheiro assembly `hello.s`. O segundo passo consiste em usar um compilador como o `gcc` ou o `clang` para gerar o executável. Para tal deverá escrever no terminal: `clang-3.8 -o hello hello.s`. O ficheiro final poderá ser executado correndo `./hello`.

1.2 Um ficheiro vazio

O ficheiro `empty.ll` contém um programa executável mínimo. O ficheiro define apenas uma função main que devolve o valor 0, não fazendo mais nada. Serve como exemplo de como se define uma função. Usa-se a sintaxe `define <tipo> @nome_da_funcao([argumentos])` seguida do corpo da função caso seja uma definição. Caso seja uma declaração (como é o caso de puts, no exemplo `hello`), o corpo não é necessário.

```
define i32 @main() {
  ret i32 0
}
```

1.3 Tipos

Uma das primeiras coisas que teremos de considerar é o tipo de dados usados. Os inteiros são representados por `iX`, sendo X o número de bits que o inteiro usa. Por exemplo: `i32` e `i64` usam 32 e 64 bits, respetivamente. Para além de inteiros existem outros tipos primitivos como `float` (32 bits) e `double` (64 bits).

Tal como em C, podemos também ter ponteiros para tipos primitivos. Como exemplos, podemos ter: `i32*` (ponteiro para um inteiro) ou `i32 (i32) *` (ponteiro para uma função que recebe um inteiro e devolve um inteiro).

Podemos ainda ter arrays, úteis para representar sequências de dados do mesmo tipo, seguindo a sintaxe `[ N x <type>]`. Por exemplo `[40 x i32]` é o tipo de um array de inteiros de tamanho 40.

1.4 Operações sobre tipos nativos

```
define i32 @main(){
```

```
  %1 = mul i32 2, 8
  %2 = add i32 %1, 2
  ret i32 %2
}
```

Neste exemplo, a função main já faz alguns cálculos. Na primeira linha guarda na variável `%1` o resultado da multiplicação da constante 2 por 8 (16). Em LLVM IR, as variáveis começam por inteiros chama-se mul e tem a sintaxe mul <tipo> op1, op2, sendo <tipo> o tipo de dados a operar e op1 e op2 os operandos. Neste caso, os operandos são os dois constantes. Para além de números, true e false são também aceites como constantes do tipo i1 (correspondente ao boolean).

Na segunda linha, voltamos a fazer uma operação (adição) usando uma constante (2) e o resultado da operação anterior (`%1`). Para além de `mul` e `add`, existem também as operações `sub` (subtracção), `udiv` (divisão inteira sem sinal), `urem` (resto da divisão inteira sem sinal), e as correspondentes operações com sinal `sdiv` e `srem`.

Em termos de operações bit a bit, temos também os operadores `and` e `or` com a idêntica sintaxe. Em termos de operações de comparação, existe o operador `icmp` que tem uma sintaxe ligeiramente mais complexa: `icmp <comp> <tipo> op1, op2`. O comparador a usar (`<comp>`) pode ser `eq` (==), `ne` (!=), `sgt` (>), `sge` (>=), `slt` (<), `sle` (<=) ou uma das versões sem sinal correspondentes (`ugt`, `uge`, `ult` e `ule`). A operação devolve `true` ou `false` conforme o resultado da comparação seja verdadeiro ou não.

Outra característica do LLVM IR é que é SSA (Static Single Assignment), o que implica que uma variável não pode ser definida duas vezes. Para testar, volte a definir `%1` antes da expressão de retorno, e veja que o código fica inválido. Apesar das variáveis poderem ter vários nomes, o código gerado costuma usar `%1`, `%2`, `%3`, etc., para expressões intermédias.

1.5 Operações sobre Arrays

```
  %arr = alloca i32, i32 3

  %ind0 = getelementptr i32, i32* %arr, i32 0
  store i32 9, i32* %ind0

  %ind1 = getelementptr i32, i32* %arr, i32 1
  store i32 8, i32* %ind1

  %ind2 = getelementptr i32, i32* %arr, i32 2
  store i32 7, i32* %ind2

  %ind1v = load i32, i32* %ind1
```

```
  ret i32 %ind1v
```

Neste exemplo, `%arr` é um array alocado dinamicamente com o tamanho 3 (apesar da constante 3 poder ser substituída pelo resultado de outra computação com `%variavel`). A primeira linha trata da alocação do espaço necessário para o array. O operador `getelementptr` (GEP) vai buscar o endereço de memória onde está guardado o elemento 0 do array `%arr`. Nas linhas seguintes irá buscar o índice 1 e 2. Em cada `store` é guardado um número (9, 8 e 7 respectivamente) no endereço de memória respectivo ao indíce.

Finalmente a instrução `load` vai buscar o valor guardado em memória no índice 1. `load` tal como `store` actuam em ponteiros de memória, que têm de ser obtidos previamente com `getelementptr`.

1.6 Operações sobre Estruturas

Uma forma de implementar estruturas e objectos é usando tipos compostos. Para este exemplo, vamos considerar um array de Java, que para além de um ponteiro para o array nativo, tem também um campo onde é guardado o número de elementos existentes. Para tal, é conveniente declarar no topo do ficheiro um alias para o tipo composto. No caso do exemplo a seguir, declaramos dois tipos, um para arrays de inteiros e outro para arrays de booleanos. Em ambos os caso, o primeiro campo é um inteiro, onde se guarda o tamanho do array.

A estrutura constrói-se através da primeira chamada a `insertvalue`. A sintaxe é `insertvalue <type> <input>, <vtype> <value>, <pos>`, o que corresponde a colocar `value` no campo `pos` do `input`. Em Java ou C, seria equivalente a `input.pos = value`, onde `type` seria o tipo da estrutura e `vtype` o tipo do campo. Caso o input seja `undef`, é criada uma nova estrutura em memória. Com chamadas encadeadas, é preenchido o elemento passo a passo, até fichar completo.

```
%IntArray = type { i32, i32* }
%BooleanArray = type { i32, i1* }
%StringArray = type { i32, i8** }

@a = global %IntArray { i32 0, i32* null}

define i32 @main() {
  %size = add i32 0, 3

  %arr = alloca i32, i32 %size
  %arr_ins = insertvalue %IntArray undef, i32 %size, 0
  %arr_ins2 = insertvalue %IntArray %arr_ins, i32* %arr, 1
  store %IntArray %arr_ins2, %IntArray* @a
```

```
  %arr_load = load %IntArray, %IntArray* @a
  %length = extractvalue %IntArray %arr_load, 0

  %store_load = load %IntArray, %IntArray* @a
  %store_arr = extractvalue %IntArray %store_load, 1
  %ind0 = getelementptr i32, i32* %store_arr, i32 0
  store i32 9, i32* %ind0

  %load_load = load %IntArray, %IntArray* @a
  %load_arr = extractvalue %IntArray %load_load, 1
  %load_ind0 = getelementptr i32, i32* %load_arr, i32 0

  %ind1v = load i32, i32* %load_ind0

  ret i32 %ind1v
}
```

A instrução extractvalue serve para ler um valor de uma estrutura, ou tipo composto. A sintaxe é `extractvalue <type> <input>, <pos>`, o que corresponde a ler `input.pos`, sendo `<type>` o tipo da estrutura a ler. No exemplo, esta instrução é usada primeiramente para obter o valor do tamanho do array, e posteriormente para obter o ponteiro para o array, a partir do qual se pode ler ou escrever no array usando `getelementptr`, `store` e `load`.

1.7 Definição e Chamada de Funções

Como já vimos, as funções podem ser definidas usando a instrução define. A seguir temos um exemplo de invocação de uma função.

```
define i32 @sum(i32 %a, i32 %b, i32 %c) {
    %1 = add i32 %a, %b
    %2 = add i32 %1, %c
    ret i32 %2
}

define i32 @main() {
  %1 = call i32 @sum(i32 1, i32 2, i32 3)
  ret i32 %1
}
```

Em primeiro lugar definimos a função `sum`, que recebe três inteiros e devolve a soma desses 3 inteiros. Como vimos anteriormente, primeiro temos de somar os dois primeiros valores

(guardando em `%1`), e só depois somamos esse resultado intermédio com o último valor e devolvemos a soma total. De notar que os argumentos são declarados com os seus tipos.

Na função `main` estamos a invocar essa função com o operador `call`. A sintaxe de chamada é `call <type> <name> (<args>)`, onde `<type>` representa o tipo de retorno da função seguido dos tipos dos parâmetros entre parêntesis. Os tipos dos parâmetros podem ser omitidos como no exemplo acima, mas não no caso de funções com número de parâmetros variável. É ainda necessário declarar o tipo de cada um parâmetros passados à função.

1.8 Instruções de fluxo de controlo

```
define i32 @main() {
entry:
  %v1 = add i32 0, 3
  %v2 = add i32 0, 2
  %ifcond = icmp eq i32 %v1, %v2
  br i1 %ifcond, label %then, label %else

then:
  %calltmp = add i32 %v1, %v2
  br label %ifcont

else:
  %calltmp1 = mul i32 %v1, %v2
  br label %ifcont

ifcont:
  %iftmp = phi i32 [ %calltmp, %then ], [ %calltmp1, %else ]
  ret i32 %iftmp
}
```

Tal como em assembly, a alteração do fluxo de controlo em LLVM IR é conseguida através de operações de salto, condicional ou incondicional. Como se pode ver no exemplo acima, a função está dividida em diferentes secções, cada uma com o seu label: `entry`, `then`, `else` e `ifcont`. O label `entry` é aquele onde a função entra por default, e coloca-se logo no início da função. Na variável `%ifcond` é guardado o valor `true` caso as variáveis `%v1` e `%v2` sejam iguais e `false` caso contrário. A instrução `br` faz saltar para o label `%then` no primeiro caso, ou para o label `%else` no segundo. A instrução `br` pode ser vista como um goto condicional dependente de uma variável.

No ramo `then` são somados os dois valores iniciais enquanto no ramo `else` estes são multiplicados. No final de cada um dos dois ramos é feito um salto (goto) através de um `br` não condicional para o label `%ifcont`, onde é usado o operador `phi` para obter o valor correcto. Este operador especial permite selecionar um valor em função do ramo percorrido pelo

programa, e é útil, por exemplo, na implementação de operadores de atribuição condicional como o operador "? :" em Java ou C.

2 Exercícios

Exercício 1 *Escrever um programa que devolva a soma dos 100 primeiros números inteiros. Para tal deverá fazer uma estrutura de controlo semelhante a um ciclo while.* □

Exercício 2 *Escrever um programa com a função de Fibonacci definida de forma recursiva.-* □

3 Soluções

Uma forma bastante simples de obter o código IR que se desconhece é escrever o código respectivo em C e usar o clang para compilar para LLVM-IR. Para tal basta executar no terminal `clang-3.8 -S -emit-llvm cheat.c`. Poderá reparar que o ficheiro `cheat.ll` resultante está cheio de metadados e atributos opcionais nos operadores, resultado de uma optimização. Poderá ignorar quase todos esses atributos e reduzir o código à forma mais elementar das expressões.

Referências

[1] LLVM Language Reference Manual `http://releases.llvm.org/3.8.1/docs/index.html`

[2] Mapping High-Level Constructs to LLVM IR `https://f0rki.gitbooks.io/mapping-high-level-constructs-to-llvm-ir/content/`

[3] LLVM Tutorial `http://releases.llvm.org/3.8.1/docs/tutorial/index.html`