7 线性 IR

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 熟悉 LLVM IR
- 会将 TeaPL 代码翻译为线性 IR
- 了解解释执行

7.1 线性 IR

本章介绍一套线性 IR 定义及其使用方式，该 IR 是 LLVM IR[1]的一个子集，可以通过 llvm 自带的 lli 工具直接解释执行。代码 7.1是一段 IR 示例。

```
@g = global i32 10 ; 声明全局变量g，类型为int32，初始值为10

define i32 @foo(i32 %0) { ; 定义函数fib，类型为i32->i32
    %x = alloca i32 ; 申请i32的栈空间，返回指针%x
    store i32 %0, i32* %x ; 将%0的值存入%x的内存单元
    %g0 = load i32, i32* @g ; 加载全局变量@g的值
    ret i32 %g0 ; 函数返回
}

define i32 @main() {
    %r0 = call i32 @foo(i32 1)
    ret i32 %r0;
}
```

代码 7.1: LLVM IR 代码示例

下面对 TeaPL 用到的 IR 指令和相关知识进行介绍。

7.1.1 类型、变量和常量

IR 支持以下类型：

- 标量类型：主要包括不同长度的有符号整数`i32`、`i8`、`i1`。
- 指针类型：以`*` 结尾的类型，如`i32*`、`i8*`。
- 数组类型：若干个同一类型的对象，如`[i32 * 2]` 表示长度为 2 的`i32` 数组。
- 自定义类型：用户可使用`type` 自己定义类型，如`%mytype = type {i32, i32}`。

变量名是一种标识符，标识符有两种基本类型：

- 局部变量：以`\%` 开头，后跟字母数字组合，如`\%r1` 或`\%1`，仅在当前函数内部有效。

[1]LLVM IR 文档: https://llvm.org/docs/LangRef.html

- 全局变量：以@ 开头，后跟字母数字组合，如\@g1 或\@1。

lli 要求 IR 中的每个变量只能定义（等号赋值）一次，该要求是为了方便后续编译器的优化操作。另外，lli 还要求如果采用纯数字编号命名变量，则必须从%0 开始递增，否则无法解释执行。

7.1.2 内存分配和数据存取

如代码 7.2所示，在函数栈帧上为局部变量分配空间可以使用alloca 指令，该指令返回指向内存单元的指针。内存分配大小以字节为单位，因此alloca 的内存大小不能是 i1。向内存单元存取数据可以分别使用store 和load 指令。

```
; <res> = alloca <type>
%x = alloca i32 ;返回指针类型：i32*
; store <ty> <value>, ptr <pointer>
store i32 1, i32* %x ; 将整数1存入\%x指向的内存
; <res> = load <ty>, ptr <pointer>
%t1 = load i32, i32* %x ; 将\%x指向内存的内容加载到\%t1
```

代码 7.2: LLVM IR 代码示例：内存分配和数据存取

注意，原代码中的变量在 IR 中都对应一块内存单元，可以使用 store-load 存取；而 LLVM IR 会引入很多的临时变量，这些变量相当于寄存器的概念。

不同的类型之间可以进行转换，如通过zext 将小数据类型转换为大数据类型（扩充 0）；通过trunc 将较大数据类型转换为小数据类型（保留低位数据）。

```
%t2 = zext i1 %t1 to i32 ; 将i1类型的%t1转换为i32类型
%t3 = trunc i32 %t2 to i8 ; 将i32类型的%t2转换为i8类型
```

代码 7.3: LLVM IR 代码示例：类型转换

数组和结构体元素的存取则涉及到寻址问题，需要使用getelementptr 指令。代码 7.4和 7.5分别是数组和结构体数据存取的例子。

```
%a = alloca [10 x i32] ;返回数组指针：[10 x i32]*
; <res> = getelementptr <ty>, ptr <ptrval>{, <ty> <idx>}*
%t1 = getelementptr *i32, [10 x i32]* %a, i32 0, i32 1
;第一个索引0对应当前数组的基地址，第二个索引1表示数组的第2个元素的偏移量。
store i32 99, i32* %t1
```

代码 7.4: LLVM IR 代码示例：数组元素存取

```
%mystruct = type { i32, i32 }
%st = alloca %mystruct
%r2 = getelementptr %mystruct, %mystruct* %st, i32 0, i32 0
store i32 1, i32* %r2
```

代码 7.5: LLVM IR 代码示例：结构体数据存取

7.1.3 算数运算

IR 中支持的算数运算主要包括add、sub、mul 和sdiv。暂不考虑无符号数运算和整数溢出的问题。

```
; <res> = add <type> <op1>, <op2>
; <op1>和<op2>的类型必须和<type>一致
```

```
%t3 = add i32 %t1, %t2 ; 加法运算：%t3 = %t1 + %t2
%t4 = sub i32 %t1, %t2 ; 减法运算：%t4 = %t1 - %t2
%t5 = mul i32 %t1, %t2 ; 乘法运算：%t5 = %t1 * %t2
%t6 = sdiv i32 %t1, %t2 ; 带符号的除法运算：：%t6 = %t1 / %t2
```

代码 7.6: LLVM IR 代码示例：算数运算

7.1.4 关系运算

IR 中支持的关系运算指令是icmp，可设置多种不同的比较模式。

```
; <res> = icmp <mod> <type> <op1>, <op2> ;mod: eq, neq, sgt, sge, slt, sle
%t3 = icmp eq i32 %t1, %t2 ; 等于
%t4 = icmp neq i32 %t1, %t2 ; 不等于
%t3 = icmp sgt i32 %t1, %t2 ; 大于
%t3 = icmp sge i32 %t1, %t2 ; 大于等于
%t3 = icmp slt i32 %t1, %t2; 小于
%t3 = icmp sle i32 %t1, %t2; 小于等于
```

代码 7.7: LLVM IR 代码示例：比较运算

7.1.5 逻辑运算

IR 中没有专门的逻辑运算指令，如逻辑非指令可以通过异或xor 指令来实现。逻辑与和或指令可以通过控制流语句实现。

```
%r2 = xor i1 %r1, true ; %r2 = !%r1
```

代码 7.8: LLVM IR 代码示例：通过异或运算实现逻辑非

7.1.6 控制流

控制流指的是程序执行时代码块之间的跳转关系。IR 中的跳转指令为br，代码块定义使用标识符和冒号（如bb1:），跳转到特定代码块时需在标识符前面加上% 分号（如br %bb1）。br 可直接跳转到目标代码块，或者在指令加入条件判断用于条件跳转。

```
bb0: ; 代码块
    %t3 = icmp eq i32 %t1, %t2
    ; br i1 <cond>, label <iftrue>, label <iffalse> ; 条件跳转
    br i1 %t3, %bb1, %bb2
bb1: ; 代码块
   ;br label <dest> ; 直接跳转
    br label %bb2
bb2: ; 代码块
    ...
```

代码 7.9: LLVM IR 代码示例：控制流

还有一条与控制流相关的条件赋值指令phi 在后面代码优化时会用到，这里暂不展开。

```
; 程序运行时如果前一个代码块是<label0>，则<res>的值是<val0>;
; 如果前一个代码块是<label1>，则<res>的值是<val1>
<res> = phi <type> [<val0>, <label0>], [<val1>, <label1>], ...
```

```
%t3 = phi i32 [%t1, %bb1], [%t2, %bb2]
```

代码 7.10: LLVM IR 代码示例：phi 指令

7.1.7 函数

定义一个函数使用`define` 语句；如果仅声明该函数则使用`declare`。在同一个 IR 文件中，不允许对同一个函数进行声明和定义。对于不同 IR 文件中的函数声明和定义，可以使用 llvm-link 工具进行链接。函数调用相关的指令主要包括调用指令`call` 和返回指令`ret`。

```
; define <type> <globalID> (<type> <v1>, <type> <v2>, ...) {...}
define i32 @foo(i32 %0) { 定义函数foo，类型是i32->i32
   ret i32 %0
}
; declare <type> <globalID> (<type> <v1>, <type> <v2>, ...)
declare void @bar(i32 %0) ; 声明函数bar，类型是i32->void
define i32 @main() {
  %r0 = call i32 @foo(i32 1)
  ret i32 %r0;
}
```

代码 7.11: LLVM IR 代码示例：函数

7.2 AST 翻译线性 IR

将 AST 翻译成 IR 代码主要分为三个步骤：

1. 遍历顶层 AST，创建全局函数/变量 IR。
2. 前序遍历每个函数的 AST，递归下降创建代码块编号和跳转关系。
3. 遍历每个代码块的指令，依次翻译代码块中的每条指令。

该翻译过程的难点在于准确关联变量的定义和使用（def-use）关系。为解决该问题，我们可以将变量的定义和使用关系限制在当前代码块内部，即强制要求变量使用前需要先 load，更新后立即 store，不允许直接使用其它代码块中 load 或计算得到的数值。代码 7.12和 7.13以阶乘函数为例展示了 TeaPL 源代码及其对应的 IR 代码。

```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

代码 7.12: TeaPL 代码

```
define i32 @foo(i32 %0) {
bb0:
    %n = alloca i32 ; 参数内存单元
```

```
    %r = alloca i32
    store i32 %0, i32* %n ; 保存参数值
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n ; 使用变量的值前先load，限制临时变量%t1仅在当前代码块使用
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r ; 使用变量的值前先load，避免与其它代码块中的%r值耦合
    %t4 = load i32, i32* %n ; 使用变量的值前先load，避免与其它代码块中的%n值耦合
    %t5 = mul i32 %t3, %t4 ; 限制临时变量%t5仅在当前代码块使用
    store i32 %t5, i32* %r ; 立即更新%r的内存单元，保证后续指令可以load到最新的数值
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n ; 立即更新%n的内存单元，保证后续指令可以load到最新的数值
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

代码 7.13: 代码 7.12对应的 IR

7.3 解释执行

线性 IR 消除了 if-else、while 等语法糖，已经非常接近汇编代码，可以从主函数入口开始依次对每条指令进行解释执行。解释执行的关键是如何保存前序指令的运行结果，使得后继指令可以使用到正确的数据。因此，与解释执行配合在一起使用的通常包括一个虚拟机，对函数的栈桢和寄存器进行模拟。解释执行和虚拟机不是本课程的重点，因此不做展开。

练习

1. 将下列 TeaPL 代码翻译为线性 IR，并使用 lli 工具进行测试。

```
let a[10]:int = {1,2,3,4,5,6,7,8,9,10};
fn binsearch(x:int) -> int {
    let high:int = 9;
    let low:int = 0;
    let mid:int = (high+low)/2;
    while(a[mid]!=x && low < high) {
        mid=(high+low)/2;
        if(x<a[mid]) {
            high = mid-1;
        } else {
            low = mid +1;
        }
    }
    if(x == a[mid]) {
        ret mid;
```

```
    }
    else {
        ret -1;
    }
}

fn main() -> int {
    let r = binsearch(2);
    ret r;
}
```

代码 7.14: TeaPL 代码