7 线性 IR

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 熟悉 LLVM IR
- 能够将 TeaPL 代码翻译为 LLVM 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展示了一段 IR 样例，包括一个简单的全局变量 `@g` 声明，以及两个函数定义`%foo` 和`%main`。

```
@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foo，类型为i32->i32，参数为%0
    %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，命名为%g0
    ret i32 %g0 ; 返回%g0
}

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详细介绍。

7.1.1 类型、变量和常量

TeaPL 用到的以下 LLVM IR 中的类型：

- 标量类型：包括不同长度的有符号整数，包括`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都是合法的标识符。局部变量仅在当前函数内部有效。另外，lli 要求如果采用纯数字编号命名局部变量，必须从%0 开始逐步递增使用数字编号，否则无法解释执行。
- 全局变量：以@开头，后跟字母数字组合，如@g1或@1。

LLVM IR 要求每个变量只能定义一次，即使用 "=" 赋值或初始化一次，因此这种 IR 又被称为静态单赋值形式。采用该形式是为了利于简化后续的编译器优化算法设计。我们会在下一章内容中详细讲解静态单赋值形式。

7.1.2 内存分配和数据存取

LLVM IR 中的内存分配主要是函数栈帧上的内存分配。如代码 7.2所示，为局部变量分配空间使用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。

```
; alloca指令形式: <ptr> = alloca <value type>
%x = alloca i32 ;返回指针类型: i32*
; store指令形式: store <value type> <value>, <ptr type> <ptr>
store i32 1, i32* %x ; 将整数1存入\%x指向的内存
; load指令形式: value = load <value type>, <ptr type> <ptr>
%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，TeaPL 原代码中的每一个变量在 IR 中都对应一块内存单元，可以使用 store-load 存取其中的数据；而 LLVM IR 会引入很多的临时变量，可以将这些变量理解为寄存器。

LLVM IR 提供了数据类型转换的指令，如通过zext..to将小数据类型转换为大数据类型（高位扩充 0）或通过trunc..to将大数据类型转换为小数据类型（仅保留低位数据）。

```
; <dst> = zext <src type> <src> to <dst type>
%t2 = zext i1 %t1 to i32 ; 将i1类型的%t1转换为i32类型的%t2
; <dst> = trunc <src type> <src> to <dst type>
%t3 = trunc i32 %t2 to i8 ; 将i32类型的%t2转换为i8类型的%t3
```

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涉及到寻址问题，需要使用getelementptr指令获取目标元素的地址后才能进行存取。代码 7.4和 7.5分别展示了数组元素和结构体域数据存取的例子。

```
%a = alloca [10 x i32] ;返回数组指针: [10 x i32]*
; <ptr> = getelementptr <element ptr type>, <array ptr type> <array>, <base type>
    <base>, <offset type> <offset>
%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
; <ptr> = getelementptr <field ptr type>, <struct ptr type> <struct>, <base type>
    <base>, <offset type> <offset>
%r2 = getelementptr %mystruct, %mystruct* %st, i32 0, i32 0
```

```
store i32 1, i32* %r2
```

代码 7.5: LLVM IR 代码示例：结构体域数据存取

7.1.3 算数运算

TeaPL 用到的 LLVM IR 算数运算指令都是有符号数的运算，包括add、sub、mul和sdiv，不涉及无符号数运算。为简化起见，我们不考虑整数运算溢出的情况。

```
; <result> = add <result type> <oprand1>, <oprand2>
; oprand1和oprand2也必须和<result 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 ; 有符号的除法运算：：%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。

```
; <result> = icmp <mod> <operand type> <oprand1>, <oprand2>
; <mod>是比较模式，包括：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

LLVM IR 中没有专门的逻辑运算指令。逻辑可以通过位运算指令xor、and、or来实现。

```
; 实现逻辑非运算：%b = !%a
; <result> = xor <type> <operand 1> <operand 2>
%b = xor i1 %a, true ;
; 实现逻辑与运算：%r = %b && %a
; <result> = and <type> <operand 1> <operand 2>
%r = and i1 %a, %b
; 实现逻辑或运算：%r = %b || %a
; <result> = or <type> <operand 1> <operand 2>
%r = or i1 %a, %b
```

代码 7.8: LLVM IR 代码示例：通过位运算实现逻辑运算

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`。代码块定义以标识符和冒号开始，如“`bb1:`”；跳转到特定代码块时需在标识符前面加上“`%`”，如“`br %bb1`”。`br`可直接跳转到目标代码块，也可以在指令中加入判断条件用于条件跳转。

```
bb0: ; 定义一个代码块
    %t3 = icmp eq i32 %t1, %t2
    ; br i1 <cond>, label <true block>, label <false block> ; 条件跳转
    br i1 %t3, %bb1, %bb2
bb1: ; 定义一个代码块
   ;br label <dst block> ; 直接跳转
    br label %bb2
bb2: ; 定义一个代码块
    ...
```

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`，下一章讲静态单赋值形式时会详细讲解，此处暂不展开。

```
; 程序运行时如果前一个代码块是<label 1>，则<result>的值是<value 2>;
; 如果前一个代码块是<label 2>，则<res>的值是<value 2>
; <result> = phi <type> [<value 1>, <label 1>], [<value 2>, <label 2>], ...
%t3 = phi i32 [%t1, %bb1], [%t2, %bb2]
```

代码 7.10: LLVM IR 代码示例：phi 指令

另外，逻辑与和逻辑或指令经常通过控制流指令以短路方式实现。

```
bb1:
  %t1 = xor i1 %a, true
  br i1 %t1, label %bb2, label %bb3
bb2:
  br label %bb3
bb3:
  %r = phi i1 [false, %bb1], [%b, %bb2]
```

代码 7.11: LLVM IR 代码示例：通过控制流指令实现%a && %b

```
bb1:
  br i1 %a, label %bb3, label %bb2
bb2:
  br label %bb3
bb3:
  %r = phi i1 [true, %bb1], [%b, %bb2]
```

代码 7.12: LLVM IR 代码示例：通过控制流指令实现%a || %b

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 LLVM IR 文件中，不允许对同一个函数既进行声明又进行定义。如果要在一个 IR 文件中调用另一个 IR 文件中定义的函数，应先在当前 IR 文件中进行声明，然后使用 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 <return type> <function ID> (<arg1 type> <arg1>, <arg2 type> <arg2>) {...}
define i32 @foo(i32 %0) { ; 定义函数foo，类型是i32->i32
   ret i32 %0
}
; declare <return type> <function ID> (<arg1 type> <arg1>, <arg2 type> <arg2>)
declare void @bar(i32 %0) ; 声明函数bar，类型是i32->void
define i32 @main() {
; <return value> = call <return type> <function ID>(<arg type> <arg value>)
  %r0 = call i32 @foo(i32 1)
; ret <return type> <return value>
  ret i32 %r0;
}
```

代码 7.13: LLVM IR 代码示例：函数声明、定义和调用

7.2 AST 翻译线性 IR

将 AST 翻译成 IR 代码的主要思路是：1）遍历顶层 AST，创建函数和全局变量的 IR；2）递归下降遍历每个函数的 AST，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。该翻译过程有两个主要难点，一是创建代码块及其跳转关系，二是关联指令参数的定义和使用（def-use）关系。

7.2.1 创建代码块及其跳转关系

LLVM 要求每个代码块必须以终结指令结束，包括`br`和`ret`。递归下降遍历 AST 时遇到以下几种情况需要创建新的代码块：

- **函数定义**：创建%bb0，添加返回指令 `ret <type> %tobeDetemined`。
- **if-else 节点**：创建三个代码块%bb-true、%bb-false 和后继代码块%bb-after。在当前代码块中添加条件跳转指令：`br i1 %tobeDetemined, label %bb-true, label %bb-false`，并将当前代码块中之前已有的终结指令转移到%bb-after 中。在%bb-true 和%bb-false 分支都添加直接调转到%bb-after 的指令。
- **while 节点**：创建三个代码块%bb-cond、%bb-body 和后继代码块%bb-after。在当前代码块中添加直接跳转到%bb-cond 的指令，并将当前代码块中之前已有的终结指令转移到%bb-after 中。在%bb-cond 添加条件跳转指令：`br i1 %tobeDetemined, label %bb-body, label %bb-after`；在%bb-body 中添加直接调转到%bb-cond 的指令。

上述思路可以完美应对 TeaPL 中的各种控制流可能，包括 while 和 if-else 嵌套的情况。实际实现时代码块编号可以采用%bb+ 编号的形式，通过维护一个计数器来实现；代码块不建议采取数字编号，容易出现编号不连贯等问题，导致无法 lli 无法执行。

7.2.2 指令参数的定义和使用

翻译 IR 时，需要确定当前指令的参数。理想情况下，应当尽可能复用已经保存在寄存器中的结果，而非再次从局部变量 load 到寄存器中。但由于该参数很可能定义自其它代码块，且可能存在多种定义，直接在翻译 IR 时解决该问题比较麻烦。因此，翻译 IR 时暂且不考虑性能问题。我们将参数的定义和使用关系限制在当前代码块内部，即要求局部变量使用前需要先 load，更新后立即 store，从而不直接使用其

它代码块中 load 或计算得到的数值。代码 7.14和 7.15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 7.14: 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 7.15: 代码 7.14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. 使用控制流指令通过短路方法改写下列代码中的逻辑与运算，并使用 lli 工具进行测试。

```
define i32 @foo(i32 %0, i32 %1) {
     %2 = alloca i32
     %3 = alloca i32
     %4 = alloca i32
     store i32 %0, i32* %3
     store i32 %1, i32* %4
     %5 = load i32, i32* %3
     %6 = load i32, i32* %4
     %7 = icmp sgt i32 %5, %6
     %8 = load i32, i32* %3
     %9 = icmp ne i32 %8, 0
    %10 = and i1 %7, %9
    %11 = zext i1 %10 to i32
    ret i32 %11
}

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

代码 7.16: LLVM IR

2. 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 7.17: TeaPL 代码

Bibliography

[1] LLVM 语言参考文档-指令部分, https://llvm.org/docs/LangRef.html#instruction-reference.