Intermediate Code Generation

静态类型检查和中间代码生成都可以通过 SDT 实现

中间代码表示

表达式的 DAG

可以为表达式构建 DAG，显示出公共子表达式

DAG 和 AST 是一样的，内部节点为运算符，子节点为运算分量，但是一个公共子表达式 N 可能会有多个父节点。构建 AST 的 SDD 修改后可以直接用来构建 DAG，只需在每次创建新节点的时候检查节点是否已存在

可以用数组存放 DAG，用下标来记录节点，这样创建节点 $< op, l, r >$ 时先检查该节点是否已存在，存在则返回其下标，否则在数组中加入该节点然后返回下标

三地址代码

地址与指令

三地址代码每条指令只有一个运算符，一般可以写成

$$x = y \text{ op } z$$

的形式。三地址代码基于两个基本概念，地址和指令

- 地址可以是名字，常量或是编译器生成的临时变量
- 指令可以有赋值，算术/逻辑运算，条件/无条件转移等

四元式与三元式

在具体实现三地址指令时，有多种描述方式，如四元式，三元式或者间接三元式

四元式有四个字段，分别是 $op, arg_1, arg_2, result$

- 单目运算符不使用 arg_2
- param 运算不使用 $arg_2, result$
- 转移指令将目标放在 $result$

三元式表示相比四元式没有 $result$ 字段，一个运算的结果 $x \text{ op } y$ 用其位置表示，但是在修改指令的位置时，引用该指令结果的所有指令都要修改，故使用**间接三元式**

间接三元式多了一个指向三元式的指针列表，使用某个指令时只需引用该列表中的指针，而修改指令位置的时候修改指针的值即可

静态单赋值

静态单赋值（SSA）是另一种中间表达形式，每个变量赋值且仅赋值一次，在控制流中使用 ϕ 函数来确定对某个变量的定值

类型和声明

类型相关的信息在中间代码生成的时候很有帮助

- 类型检查：检查运算分量的类型和运算符所期待的类型是否匹配
- 翻译时的应用：为某个变量分配内存空间时需要其类型信息。虽然实际的存储空间分配是在运行时完成的，但编译时可以预先确定相对偏移

类型表达式

可以用类型表达式来表示类型本身的结构，类型表达式基于如下的定义

- 基本类型是一个类型表达式
- 类名是一个类型表达式
- 类型构造算子运用于类型上后可以得到一个类型表达式
    - array[数字，类型表达式] 用于表示数组
    - record[(字段，类型表达式)的列表] 用于表示类似结构体的由有名字段组成的类型
    - $\rightarrow$ 用于表示函数类型
    - 如果 s, t 是类型表达式，则其笛卡尔积 $s \times t$ 也是类型表达式，用于表达函数的参数
- 类型表达式可以包含取值为类型表达式的变量

类型等价

类型等价可以分为名等价和结构等价

对于结构等价，两个类型等价 $\iff$ 以下某个条件成立

- 它们是相同的基本类型
- 它们是相同的类型构造算子应用于结构等价的类型
- 一个类型是另一个类型表达式的名字

而对于名等价来说，没有最后一个条件，类型的名字仅代表其自身

局部变量的存储布局

根据变量的类型即可确定变量需要的内存。由于局部变量总是分配到连续的内存（暂不考虑对齐的问题），故可以为每个变量分配一个相对于该区间开始位置的相对地址，而对于可变大小的数据结构，为其分配指针的空间即可

中间代码生成

表达式的翻译

表达式可以采取增量式的翻译方案

对于数组元素的寻址，一个 k 维数组 $A[n_1][n_2]\dots[n_k]$ 如果元素大小为 w，则寻找元素 $A[i_1][i_2]\dots[i_k]$ 的位置为

$$base+(i_1\times\prod_{i=2}^{k}n_i+i_2\times\prod_{i=3}^{k}n_i+\cdots+i_k)\times w$$

可以将这个计算与具体的文法关联起来

类型检查

为了类型检查，编译器需要给源程序每一个部分赋予一个类型表达式，然后确定这些类型表达式是否满足一组逻辑规则，即源语言的类型系统

类型检查有两种形式

- 类型综合（type synthesis）：根据子表达式的类型构造出表达式的类型，要求名字先声明再使用，如两个 int 的和的类型是 int
- 类型推导（type inference）：根据语言结构的使用方式来确定该结构的类型，如 null 是测试列表是否为空的函数，则 null(x) 要求 x 必须是列表类型

编译器也可以进行隐式的类型转换，转换规则有拓宽和窄化，拓宽可以保留信息，而窄化可能会丢失信息，编译器能进行的类型转换一般只能是拓宽转换，窄化转换需要程序员手动完成

针对函数重载的问题，一般根据参数的类型确定其含义

控制流

控制流语句的翻译常常是与布尔表达式的翻译结合在一起的。

布尔表达式

布尔表达式常用来

- 改变控制流
- 计算逻辑值

这两种应用方式常常要根据上下文决定

考虑由如下文法生成的布尔表达式

$$B \to B||B \mid B\&\&B \mid !B \mid (B) \mid E \text{ rel } E \mid true \mid false$$

如果程序语言不要求对布尔表达式的各个部分求值，则可以根据短路规则优化布尔表达式的求值过程，只要**已经求值的部分能确定整个表达式的值**

如代码 `if (x < 100 || x > 200 && x != y) x = 0;` 可以被翻译为

```
    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1:
```

控制流语句

考虑文法

$$\begin{aligned} S &\to \text{if } (B) S_1 \\ S &\to \text{if } (B) S_1 \text{ else } S_2 \\ S &\to \text{while } (B) S_1 \end{aligned}$$

其中 B 和 S 有综合属性 $code$ 代表生成的三地址代码，B 有继承属性 $true$ 和 $false$ 代表为真/为假时跳转的目标，S 有继承属性 $next$ 代表语句结束后跳转的位置

则对于控制流语句，其 SDT 为

产生式	语义规则
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if} (B) S_1 \textbf{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}' S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow \textbf{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

同样的，对于布尔表达式，有 SDT

产生式	语义规则
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \;\mathbf{rel}\; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\mathtt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\mathtt{goto}'\ B.true)$ $\mid\mid gen('\mathtt{goto}'\ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\mathtt{goto}'\ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\mathtt{goto}'\ B.false)$

避免冗余的 goto

可以利用指令自然地流动到下一条指令的特性，添加一个特殊的标号 fall 表示不生成跳转语句，如对 if 语句翻译的 SDT 中将 $B.true$ 设为 fall

布尔表达式用于计算逻辑值

可以统一处理程序中出现的布尔表达式，但是使用不同的函数用来生成控制流或求值的代码。

对于求值的情况，可以先为布尔表达式生成跳转代码，然后在其跳转代码的 true/false 出口将 true/false 赋给临时变量

回填

在布尔表达式的翻译中，需要确定跳转语句的目标，解决这个问题的方法是使用继承属性 next 确定跳转的目标，如果不使用继承属性，可以采用回填的策略

回填策略将一个跳转指令的列表作为综合属性传递，同一个列表中的跳转指令有相同的跳转目标，即生成不完整的跳转指令，然后在跳转目标能确定的时候将这些标号填回不完整的跳转指令

引入两个综合属性 $truelist$ 和 $falselist$

同时引入辅助函数

- $Makelist(i)$：创建一个只含有 i 的列表
- $Merge(p_1, p_2)$：合并两个列表
- $Backpatch(p, i)$：将 i 作为跳转目标插入 p 中的指令

则对于布尔表达式和控制流语句，可以采用如下的方案翻译

1)	$B \to B_1$ `||` M B_2	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \to B_1$ `&&` M B_2	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \to$ `!` B_1	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \to$ `(` B_1 `)`	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
6)	$B \to$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
7)	$B \to$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$M \to \epsilon$	{ $M.instr = nextinstr$; }

1) $S \to$ **if** `(` B `)` $M\ S_1$ { $backpatch(B.truelist,\ M.instr)$; $S.nextlist = merge(B.falselist,\ S_1.nextlist)$; }

2) $S \to$ **if** `(` B `)` $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$; $backpatch(B.falselist,\ M_2.instr)$; $temp = merge(S_1.nextlist,\ N.nextlist)$; $S.nextlist = merge(temp,\ S_2.nextlist)$; }

3) $S \rightarrow$ **while** M_1 (B) M_2 S_1
$\{\ backpatch(S_1.nextlist,\ M_1.instr);$
$backpatch(B.truelist,\ M_2.instr);$
$S.nextlist\ =\ B.falselist;$
$gen('\texttt{goto}'\ M_1.instr);\ \}$

4) $S \rightarrow \{\ L\ \}$ $\{\ S.nextlist\ =\ L.nextlist;\ \}$

5) $S \rightarrow A\ ;$ $\{\ S.nextlist\ =\ \textbf{null};\ \}$

需要引入新的非终结符 M 来记录下一条指令的位置，引入终结符 N 生成 goto 指令

break 与 continue 的处理

break 与 continue 都与外围的语句相关，如对于 break 可以按照如下的方式处理

- 跟踪外围语句 S
- 生成不完整的跳转语句
- 将该语句加入 $S.nextlist$

continue 的思想类似，只是加入的位置不同