# 指令式语言转函数式语言

高级编译课程项目

崔 嵬 1301214236

秦若然 1301214259

2013.12

目录

[指令式语言转函数式语言 0](#_Toc376370901)

[一、 项目背景 1](#_Toc376370902)

[（一） 选题原因 1](#_Toc376370903)

[（二） 项目用途 1](#_Toc376370904)

[（三） 逻辑转换过程带来的其他好处 1](#_Toc376370905)

[二、 分析工作 2](#_Toc376370906)

[（一） 指令式语言与函数式语言相互转换的难易度分析 2](#_Toc376370907)

[1. 函数式语言 -> 指令式语言 2](#_Toc376370908)

[2. 指令式语言 -> 函数式语言 2](#_Toc376370909)

[3. 局部变量/全局变量 -> 仅局部变量 2](#_Toc376370910)

[（二） 针对输入与输出的分析 2](#_Toc376370911)

[1. 输入分析：实现时对C语法进行约束 —— CSC 2](#_Toc376370912)

[2. 输出之一分析：标准数学函数形式 3](#_Toc376370913)

[3. 输出之一分析：ERLANG函数式语言形式 4](#_Toc376370914)

[三、 转换算法设计 4](#_Toc376370915)

[（一） 标准数学函数：高级语言到高级语言 4](#_Toc376370916)

[1. 尾递归转换法消除循环 4](#_Toc376370917)

[2. 路径敏感分析消除条件分支 5](#_Toc376370918)

[3. 表达式代入法消除重赋值 5](#_Toc376370919)

[（二） 函数式语言目标：高级语言到中间表示到高级语言 7](#_Toc376370920)

[1. 课程内容：CSC的修改/划分基本块/构造流图/构造Call Graph 7](#_Toc376370921)

[2. DFA拓展设计与跨过程应用：活跃全局变量分析(IPA/DFA-Ex) 10](#_Toc376370922)

[3. 课程内容：SSA(DFA/DOM/IDOM/DF-Set/Add-Phi/Add-Index) 14](#_Toc376370923)

[4. 中间表示到函数式语言的最终转化：有限状态机转换法 16](#_Toc376370924)

[5. 细节部分的映射解决方案：数组转换与标准控制台I/O 18](#_Toc376370925)

[四、 相关证明 18](#_Toc376370926)

[1. 转换完备性：证明仅实现CSC到函数式语言的转换是完备的 18](#_Toc376370927)

[2. 活跃全局变量分析：过程对全局变量定值算法的正确性 24](#_Toc376370928)

[3. 活跃全局变量分析：过程对全局变量使用算法的正确性 25](#_Toc376370929)

[五、 总结 26](#_Toc376370930)

[六、 附实例测试 27](#_Toc376370931)

[（一） 标准数学函数实例测试 27](#_Toc376370932)

[（二） Erlang函数式语言实例测试 28](#_Toc376370933)

1. 项目背景
   1. 选题原因

高级编译的课程项目的主要是对课上学到的编译优化、分析等算法进行实践：在把C代码转换为三地址指令后，通过一些列的优化处理得到质量更好的代码，最终转换回C代码。但考虑到自身项目的相关性和有用性，因此决定对处理过的三地址指令向函数式语言进行转换。

* 1. 项目用途

自身所从事的项目以分布式计算为主，分布式云系统采用了Erlang实现，要达到与该系统软件最佳的原生性和兼容性，就要求用户采用Erlang语言进行map/reduce或者pregel compute等逻辑的编程，而毕竟由于这类函数式语言对于大部分用户来说难以接受，或者难以写出十分高效的算法逻辑，因此，此项目将作为该云系统的一个附加组件，可选地为那些只接受指令式编程逻辑的用户进行代码转换，免除其学习理解函数式语言。

* 1. 逻辑转换过程带来的其他好处

为实现同样一个循环逻辑，函数式语言不得不采用递归，此时需要用户进行抉择，到底是写成非尾递归地简易形式，还是写成尾递归的复杂形式？

a.非尾递归地简易形式：

f(1) -> 1;

f(2) -> 1;

f(N) -> f(N-1)+f(N-2).

b.尾递归的复杂形式：

f(1) -> 1;

f(2) -> 1;

f(N) -> g(1,1,N).

g(A,\*,2) -> A;

g(A,B,N) -> g(A+B, A, N-1).

可见，两者各有优缺点，但对于用户来说，更倾向于前者(a)，虽然这会带来巨大的计算时间代价。本项目转换保证了时间复杂度的量级与指令式源代码相当。这样，既不需要用户自行实现复杂的(b)形式，又保证了代码执行的时间复杂度与复杂的(b)形式是量级相当的。

1. 分析工作
   1. 指令式语言与函数式语言相互转换的难易度分析

简单分析可知，函数式语言是指令式语言的子集，也即函数式语言相对于指令式语言具有更多的约束，而指令式语言对函数式语言的语法可以直接实现。因此，显然从函数式语言到指令式语言的转换简单，而反之是一个较为困难的问题。

具体理由如下：

* + 1. 函数式语言 -> 指令式语言

a) 单赋值语法系统显然是可以被可重赋值语法系统所直接接受的；

b) 任何函数式语言的尾递归都可以直接转换为指令式语言的尾递归，以gcc为例，在编译器开启-O2优化之后，可以完全把尾递归优化为循环递推形式。

* + 1. 指令式语言 -> 函数式语言

a) 可重赋值语法系统转换到单赋值语法系统，需要进行额外的单赋值处理；

b) 指令式语言的循环在函数式语言系统中不存在，只能借助递归实现，同时，考虑到执行效率、空间占用情况，产生的新递归函数应该都是尾递归形式。

* + 1. 局部变量/全局变量 -> 仅局部变量

这是C语言转Erlang语言所带来的额外困难问题之一，同样需要解决如何使用子集语法系统实现全集的任何语法的问题。从这一点可以说明，从仅支持局部变量的语言转换到局部变量/全局变量的语言简单，反之较难。

* 1. 针对输入与输出的分析
     1. 输入分析：实现时对C语法进行约束 —— CSC

考虑到C语法系统十分庞大，短时间内无法完备支持，所以借助对CSC源码的参考，仅实现了其语法系统的一个子集，可以证明，该子集对于实现算法逻辑而言是完备的（也即任意全局语法都可等价转换为子集内部的语法）。

利用CSC所提供的语法子集已有的语法支持如下：

1）过程无返回值

2）一维数组（全局、局部均可）

3）循环语句： while..

4）条件语言：if .. / if .. else ..

5）无 break、continue、return

* + 1. 输出之一分析：标准数学函数形式

所谓标准数学函数，是指：

* 一个或多个独立分支；
* 每个独立分支由一个超级条件和超级表达式构成；
* 不同分支之间超级条件合并为全集、且两两无交叉。

例如：

f(X) = { 1, when (X == 0)

{ f(X / 2) + 2, when (X > 0 && X % 2 == 0)

{ f(X + 3) + 1, when (X > 0 && X % 2 != 0)

这种表示方法的优点主要有以下两点：

（1）最通用的表示法，适用于任意函数式目标语言。

（2）符合数学直观逻辑，易于接受理解，易于分析。

（3）不存在死代码问题，常量是自然传播的。

当然，它也具有一些缺点。对程序执行来讲，计算拓扑信息丢失，超级条件之间存在不可避免、不可能消除的公共子表达式。

如果一份C代码的某个过程具有很少的条件分支语句，那么这种表示方法极为合适。

* + 1. 输出之一分析：ERLANG函数式语言形式

以下为ERLANG函数式语言目标的样例：

f(X) = f(X, 0).

f(X, Ans) = case X of

0 -> Ans + 1,

\_ -> case X % 2 of

0 -> f(X / 2, Ans + 2),

1 -> f(X + 3, Ans + 1)

end

end.

可以看出，与标准数学函数形式相反，ERLANG函数式语言的存在语句的拓扑执行顺序，但其缺点是直观性较差，也可能不能直接转换到另一种函数式语言。

由于适用范围、优缺点的不同，两种输出形式均有一定的使用意义。后面将分别讨论两个方向的转换算法，必要时对算法正确性的进行证明。

1. 转换算法设计
   1. 标准数学函数：高级语言到高级语言

进行该转换主要分为以下三个步骤：

消除循环(尾递归)-> 消除条件(路径敏感分析)-> 消除赋值(表达式代入)

* + 1. 尾递归转换法消除循环

一个过程看作转换的基本单位，那么过程所对应的流图可能是任意有向图，这是由于循环导致的。而一个循环的多次执行意味着该循环内的赋值语句可能被执行多次，而且每次赋以不同的值，违背了函数式语言的单赋值要求。通过尾递归转换法能够对任意while语句以一种通用的规则转换为非循环的递归形式，而且引入递归都属于尾递归。

|  |  |
| --- | --- |
| f(..) {  <Any-X>;  while (C)  Body;  <Any-Y>;  } | f(..) ->  <Any-X>,  { NewDef } = f\_0( UseDef ),  <Any-Y>.  f\_0(UseDef) -> case C of  true -> Body, f\_0(NewUseDef);  false -> Def  end. |

尾递归转换法的核心思想是，把循环视为一个过程，前一次Body执行后的最新变量值为下一次Body执行前的形参输入。子过程调用在函数式语言中是一个与赋值语句等地位的计算单位，此时过程对应的流图由任意有向图转换为有向无环图。

* + 1. 路径敏感分析消除条件分支

只有先完成前面的循环消除，才能通过路经敏感分析进行不同分支的分析，这是因为存在循环的流图具有无限条路径，而循环消除后，形成的有向无环图的路径数量是有限的（路径总数与过程中if数量有关）。

抽取每条路径的执行序列，每个路径序列将会对应一个独立分支。

y = floor(x); | 路径1： | 路径2：

if (y<0) | y = floor(x) | y = floor(x)

y = -y; | ASSERT y < 0 | ASSERT !(y < 0)

x = sqrt(y); | y = -y | x = sqrt(y)

return x; | x = sqrt(y) | RET x

| RET x |

关于路径分支总数的计算问题：考虑一个已经消除循环的过程，如果其包含了C个if条件分支语句，那么路径总数至少为C+1条（分支是逐层嵌套的），至多为2^C条（分支是同层次序列化的）。这就是为什么该转换方法更适合条件分支较少的过程。考虑一般情况下C的值不会太大，因此路径总数往往是可以接受的。

* + 1. 表达式代入法消除重赋值

对于每条路经，维护一个从var\_name:string到var\_expr:string的映射，var\_name来源于所有Use、Def变量，初始化每个var\_expr=var\_name。自上而下进行Use代入和Def更新两个过程。

算法描述：

|  |
| --- |
| foreach (varname : USE union DEF)  map[varname] = varname;  foreach (command : procedure\_path) {  foreach (use\_var : command.use\_set())  command = command.replaceAll(use\_var, map[use\_var]);  def\_var = command.def();  map[def\_var] = command;  } |

以路径1为例：

y = floor(x)

ASSERT y < 0

y = -y

x = sqrt(y)

RET x

初始化变量和表达式的映射关系为：

|  |  |  |
| --- | --- | --- |
| 变量 | x | y |
| 表达式 | x | y |

当完成所有代入、更新后，最终可以得到：

y = floor(x) |

ASSERT floor(x) < 0 | when floor(x) < 0

y = -floor(x) |

x = sqrt(-floor(x)) |

RET sqrt(-floor(x)) | f(x) = sqrt(-floor(x))

此时对应的变量和表达式的映射关系为：

|  |  |  |
| --- | --- | --- |
| 变量 | x | y |
| 表达式 | sqrt(-floor(x)) | -floor(x) |

对每条路径进行表达式代入，就能得到一个过程所有独立分支构成的标准数学函数:

y = floor(x); |

if (y<0) | f(x) = { sqrt(-floor(x))

y = -y; | { when floor(x) < 0

x = sqrt(y); | { sqrt(floor(x))

return x; | { when !(floor(x) < 0)

至此，完成到“标准数学函数”的转换。

表达式代入法的好处是不仅不存在重赋值的问题，而且自然地消除了死代码（死代码一定不存在于超级表达式中）、自然地进行了常量传播（没有变量，所有初始值都被尽可能地向后代入传播）。

* 1. 函数式语言目标：高级语言到中间表示到高级语言

因为该转换过程存在中间表示（三地址指令）的辅助，所以一部分处理过程与编译课程所学到的算法更相关，其主要步骤概括如下：

i) CSC的修改/划分基本块/构造流图/构造Call Graph ->

ii) 活跃全局变量分析(DFA/IPA/DFA-Ex) ->

iii) SSA(DFA/DOM/IDOM/DF-Set/Add-Phi/Add-Subscript) ->

iv) 有限状态机转换法 ->

v) 数组转换方案、内建函数转换标准控制台I/O方案

* + 1. 课程内容：CSC的修改/划分基本块/构造流图/构造Call Graph

CSC需要进行修改的部分主要包括了以下五个部分：

1）把压栈式实参输入多指令修改为单句函数调用指令

2）把数组下标算地址多指令修改为单句[]=和=[]指令

3）把条件判断、转移双指令修改为单句判断转移指令

4）增加全局变量、数组和局部变量、数组的声明指令

后面我们将不断以fabbo为例，说明转换算法的中间调试信息，输入如下：

auto ans;

void fabbo(auto n) {

auto i, a, b;

if (n<=2) {

ans=1;

} else {

a=1;

b=1;

while (n>=3){

ans=a+b;

a=b;

b=ans;

n=n-1;

}

}

}

void main() {

auto n;

scan(n);

fabbo(n);

print(ans);

println();

}

按照课程内容介绍，将以上代码指令化，转换为三地址指令：

# -------- Global Variable -----------

# \_ans,

# -------- Procedure Heading --------

# ProcName: fabbo

# FormalArgs: \_n,

# LocalVar: \_b, \_a, \_i,

# -------- Classify Instruction ------

# 0: T JP if \_n > 2 goto 3

# 1: T RD \_ans = 1

# 2: F JP goto 13

# 3: T RD \_a = 1

# 4: F RD \_b = 1

# 5: T JP if \_n < 3 goto 13

# 6: T RD T6 = \_a + \_b

# 7: F RD \_ans = T6

# 8: F RD \_a = \_b

# 9: F RD \_b = \_ans

# 10: F RD T10 = \_n - 1

# 11: F RD \_n = T10

# 12: F JP goto 5

# 13: T EL ret

# -------- Procedure Heading --------

# ProcName: main

# FormalArgs:

# LocalVar: \_n,

# -------- Classify Instruction ------

# 0: T RD T0 = read

# 1: F RD \_n = T0

# 2: F CL call fabbo | \_n

# 3: F EL write \_ans

# 4: F EL writeln

# 5: F EL ret

按照课程内容介绍，划分基本块为：

# block 1 {

# \* T JP if \_n > 2 goto 3

# } line: (0,1);

# block 2 {

# \* T RD \_ans = 1

# \* F JP goto 13

# } line: (1,3);

# block 3 {

# \* T RD \_a = 1

# \* F RD \_b = 1

# } line: (3,5);

# block 4 {

# \* T JP if \_n < 3 goto 13

# } line: (5,6);

# block 5 {

# \* T RD T6 = \_a + \_b

# \* F RD \_ans = T6

# \* F RD \_a = \_b

# \* F RD \_b = \_ans

# \* F RD T10 = \_n - 1

# \* F RD \_n = T10

# \* F JP goto 5

# } line: (6,13);

# block 6 {

# \* T EL ret

# } line: (13,14);

# block 1 {

# \* T RD T0 = read

# \* F RD \_n = T0

# \* F CL call fabbo | \_n

# \* F EL write \_ans

# \* F EL writeln

# \* F EL ret

# } line: (0,6);

按照课程内容介绍，构造Flow Graph 和 Call Graph，如下：

1. (b) (c)
   * 1. DFA拓展设计与跨过程应用：活跃全局变量分析(IPA/DFA-Ex)

不同于其他函数式语言，Erlang函数式语言无全局变量，仅靠SSA解决语法转换的中间表示还不够，重要的问题之一是把全局数据转换为等价的局部数据。

为解决该问题，需要进行一个跨过程的活跃全局变量分析。

那么，如何把全局变量局部化？依照以下两个原则：

i) 一个过程如果Use了某个全局变量，则需要把该全局变量作为过程的形参传入；

ii) 一个过程如果Def了某个全局变量，则需要该过程在Exit时把该全局变量的最新值作为返回值之一传出；

值得注意的是：

i) 任何调用子过程的指令，都需要把最新的全局变量值作为实参添加到调用形参表的尾部；同时，需要获知在子过程执行结束时通过返回值Def了哪些全局变量；

ii) 一个子过程调用指令，可以视为一个多值赋值语句，Use分为局部Use和全局Use，Def一定是全局Def。

首先，我们来确定子过程Exit时需要返回哪些全局变量新值：

如果一个过程f没有子过程调用，那么它的所有全局变量定值都来自于赋值语句，此时可以通过一次基本块内扫描“立即”确定该过程对全局变量子集的“定值”情况。但是，如果f存在子过程调用g，情况不容乐观，因为此时g产生的“内部”定值可能尚不清楚；更可能的是，即使先去求g，仍然可能存在这个问题，比如g又调用了f (间接递归)。

考虑到定值在过程间具有传递性，因此我们可以构造如下就全局变量定值的数据流分析中。

作用图：Call Graph

A=

B=

A=

B=

方向：逆向

Def集：过程内部已确定的所有全局定值(动态)

求Def时不确定的子过程定值初始化：∅

传递函数：IN = OUT ∪ Def\*

B=

汇合运算：OUT = ∪ INsucc

例如：

g() { y = x; }

f() {

x = 1;

g(); // 初始不确定

x = y + 1;

}

初始化：

Def[f]={x}

Def[g]={y}

迭代后：

Def[f]={x, y}

Def[g]={y}

确定子过程Entry时需要传入的全局变量新值：

**传入变量集 = Entry处所有活跃变量 ∩ 全局变量**

求传入变量集可以与前述定值做法一样，但不是最优的。比如

auto n;

f() { n = -n; }

main() {

n=10;

f();

}

上例子中，f过程入口需要n，main过程调用了f。但由此推知main过程入口需要n是不正确的。事实上，main过程入口并不需要n。在全局变量的活跃变量数据流分析中，数据流分析粒度不得不细化到过程内部。所以我们采用两层迭代：反复进行过程内活跃变量分析，内层只要有一个迭代不收敛，外层迭代就不能收敛。

作用图：Intra-Procedure Flow Graph

方向：逆向

Use集：过程内已确定的所有全局活跃变量(动态)

求Use时不确定的子过程变量初始化： ∅

传递函数：IN = (OUT - Def )∪ Use\*

汇合运算：OUT = ∪ Insucc

while (至少一个过程迭代产生新活跃变量) {

foreach (单个过程proc) {

重新计算proc的最新GEN集；

对于进行过程内活跃变量分析；

}

}

基于上述算法，我们就进一步精确了传入变量集合元素，例如：

|  |  |
| --- | --- |
| g() { y = x; }  f() {  x = 1;  g(); // 初始不确定  x = y + 1;  }  初始化：  Use[f]={} // 没有y  Use[g]={x}  迭代后：  Use[f]={}  Use[g]={x} | g() { y = x; }  f() {  ~~x = 1;~~  g(); // 初始不确定  x = y + 1;  }  初始化：  Use[f]={} // 没有y  Use[g]={x}  迭代后：  Use[f]={x}  Use[g]={x} |

最后划分基本块后的代码经过活跃全局变量使用与定制，可转换为：

# -------- DFA Proc Glob Live Out --------

# fabbo.glob-def : \_ans

# main.glob-def : \_ans

# --- Proc fabbo Current Use-Def Set ---

# B[0].semi-use = {}; B[0].semi-def = {}.

# B[1].semi-use = {}; B[1].semi-def = {}.

# B[2].semi-use = {}; B[2].semi-def = {\_ans }.

# B[3].semi-use = {}; B[3].semi-def = {}.

# B[4].semi-use = {}; B[4].semi-def = {}.

# B[5].semi-use = {}; B[5].semi-def = {\_ans }.

# B[6].semi-use = {}; B[6].semi-def = {}.

# B[7].semi-use = {}; B[7].semi-def = {}.

# --- DFA Proc fabbo Current Glob Live In ---

# \_ans,

# --- Proc main Current Use-Def Set ---

# B[0].semi-use = {}; B[0].semi-def = {}.

# B[1].semi-use = {\_ans }; B[1].semi-def = {\_ans }.

# B[2].semi-use = {}; B[2].semi-def = {}.

# --- DFA Proc main Current Glob Live In ---

# \_ans,

# --- Proc fabbo Current Use-Def Set ---

# B[0].semi-use = {}; B[0].semi-def = {}.

# B[1].semi-use = {}; B[1].semi-def = {}.

# B[2].semi-use = {}; B[2].semi-def = {\_ans }.

# B[3].semi-use = {}; B[3].semi-def = {}.

# B[4].semi-use = {}; B[4].semi-def = {}.

# B[5].semi-use = {}; B[5].semi-def = {\_ans }.

# B[6].semi-use = {}; B[6].semi-def = {}.

# B[7].semi-use = {}; B[7].semi-def = {}.

# --- DFA Proc fabbo Current Glob Live In ---

# \_ans,

# --- Proc main Current Use-Def Set ---

# B[0].semi-use = {}; B[0].semi-def = {}.

# B[1].semi-use = {\_ans }; B[1].semi-def = {\_ans }.

# B[2].semi-use = {}; B[2].semi-def = {}.

# --- DFA Proc main Current Glob Live In ---

# \_ans,

# -------- DFA Proc Glob Live In (Final) --------

# fabbo : \_ans

# main : \_ans

完成活跃全局变量分析之后：

void f(args); 转化为了 y & z = f(args ∪ x & y); 的形式

* + 1. 课程内容：SSA(DFA/DOM/IDOM/DF-Set/Add-Phi/Add-Index)

该算法采用了正向数据流分析和若干深搜流图、支配树等相关课程内容相关的算法，具有方法参考源代码，这里不再赘述，仍然就fabbo样例，我们得到以下结果：

# -------- Dominate Relation --------

# 0 idom 1; 1 idom 2; 1 idom 3; 3 idom 4; 4 idom 5; 1 idom 6; 6 idom 7;

# -------- Dominator Frontier --------

# # DF(0) =

# DF(1) =

# DF(2) = 6,

# DF(3) = 6,

# DF(4) = 4, 6,

# DF(5) = 4,

# DF(6) =

# DF(7) =

# -------- Insert Phi Function --------

# B0 +=

# B1 +=

# B2 +=

# B3 +=

# B4 += \_a, \_ans, \_b, \_n,

# B5 +=

# B6 += \_a, \_ans, \_b, \_n,

# B7 +=

插入phi函数添加下标，转换为：

# ------ Instruction after Modify Subscript ------

# block 1 {

# \* T JP if Id0\_n > 2 goto 3

# }

# block 2 {

# \* T RD Id1\_ans = 1

# \* F JP goto 6

# }

# block 3 {

# \* T RD Id2\_a = 1

# \* F RD Id2\_b = 1

# \* T JP goto 4

# }

# block 4 {

# ! Id3\_a = phi(3:Id2\_a, 5:Id4\_a)

# ! Id3\_ans = phi(3:Id0\_ans, 5:Id4\_ans)

# ! Id3\_b = phi(3:Id2\_b, 5:Id4\_b)

# ! Id2\_n = phi(3:Id0\_n, 5:Id3\_n)

# \* T JP if Id2\_n < 3 goto 6

# }

# block 5 {

# \* T RD T6 = Id3\_a + Id3\_b

# \* F RD Id4\_ans = T6

# \* F RD Id4\_a = Id3\_b

# \* F RD Id4\_b = Id4\_ans

# \* F RD T10 = Id2\_n - 1

# \* F RD Id3\_n = T10

# \* F JP goto 4

# }

# block 6 {

# ! Id1\_a = phi(2:Id0\_a, 4:Id3\_a)

# ! Id2\_ans = phi(2:Id1\_ans, 4:Id3\_ans)

# ! Id1\_b = phi(2:Id0\_b, 4:Id3\_b)

# ! Id1\_n = phi(2:Id0\_n, 4:Id2\_n)

# \* T EL ret

# }

重命名时，需要注意的是对于返回多参数的子过程调用，可以视为一条指令同时对多个变量进行了定值（类似多条赋值指令），比如：

y & z = f(args ∪ x & y); 转化为 y2 & z1 = f(args ∪ x1 & y1);

* + 1. 中间表示到函数式语言的最终转化：有限状态机转换法

采用有限状态机转换法，对于任意有向流图都适用（如存在环的流图）。

每个基本块将被转化为一个子函数，函数的入口形参表为：活跃局部变量 + 活跃全局变量；函数的返回值集为：全局变量定值。

该转换方法的思想是，每个子函数（当前状态）利用传入的所需活跃变量进行基本块内计算，完成计算后把对外部产生影响的新定值传给下一个函数（下一个状态）。

另外，活跃性处理和SSA处理后的中间表示，每条指令都能直接映射到Erlang函数式语言。这些指令包括：赋值映射、数组访问、条件/无条件转移、函数调用。

最终，我们将样例代码转化为Erlang编译可通过、执行正确的函数式语言源代码：

-module(fabbo).

-export([entry\_main/0, fabbo/2, main/1]).

fabbo(\_n, \_ans) ->

fabbo\_b1(\_n, \_ans).

fabbo\_b1(Id0\_n, \_ans) ->

case (Id0\_n > 2) of

true -> fabbo\_b3(Id0\_n, \_ans);

false -> fabbo\_b2()

end.

fabbo\_b2() ->

Id1\_ans = 1,

fabbo\_b6(Id1\_ans).

fabbo\_b3(\_n, \_ans) ->

Id2\_a = 1,

Id2\_b = 1,

fabbo\_b4(\_n, \_ans, Id2\_a, Id2\_b).

fabbo\_b6(Id2\_ans) ->

{Id2\_ans}.

fabbo\_b4(Id2\_n, Id3\_ans, Id3\_a, Id3\_b) ->

case (Id2\_n < 3) of

true -> fabbo\_b6(Id3\_ans);

false -> fabbo\_b5(Id2\_n, Id3\_a, Id3\_b)

end.

fabbo\_b5(Id2\_n, Id3\_a, Id3\_b) ->

T6 = Id3\_a + Id3\_b,

Id4\_ans = T6,

Id4\_a = Id3\_b,

Id4\_b = Id4\_ans,

T10 = Id2\_n - 1,

Id3\_n = T10,

fabbo\_b4(Id3\_n, Id4\_ans, Id4\_a, Id4\_b).

main(\_ans) ->

main\_b1(\_ans).

main\_b1(Id0\_ans) ->

T0 = list\_to\_integer(lists:delete(10,io:get\_line([]))),

Id1\_n = T0,

{Id1\_ans} = fabbo(Id1\_n, Id0\_ans),

io:put\_chars(integer\_to\_list(Id1\_ans)),

io:put\_chars([10]),

{Id1\_ans}.

可以看出，状态间的转换仍然是尾递归的，属于间接的尾递归。

* + 1. 细节部分的映射解决方案：数组转换与标准控制台I/O

关于数组转换方案、内建函数转换标准控制台I/O方案

1）数组转换方案：

A = C[D] -> A =[] C D -> A = element(D, C)

C[D] = A -> C []= C D A -> C2 = setelement(D, C, A)

2）通过内建函数实现标准控制台I/O方案的转换：

scan(N) -> N=list\_to\_integer(lists:delete(10,io:get\_line([])))

print(N) -> io:put\_chars(integer\_to\_list(N))

println() -> io:put\_chars([10])

1. 相关证明
   * 1. 转换完备性：证明仅实现CSC到函数式语言的转换是完备的

令全集语法系统为G，子集语法系统与G'。与全集语法的等价性证明，只需要构造一种时间复杂度可接受的映射函数f，使得：

任意g∈ (G - G')，均有 f(g) = g'∈ G'。

下面是映射函数f的参考：

1）“无返回值过程”实现任意“有返回值过程”：

利用全局变量做返回值的存储空间。

|  |  |
| --- | --- |
| Type f(Args) {  ...  return Value;  ...  } | Type f\_Ret;  void f(Args) {  ...  {f\_Ret = Value; return;}  ...  } |

2）“一维数组”实现任意“多维数组”：

以三维数组为例，定义A[**N,M,S**] 访问A[I,J,K]，则对应的一位数组操作方式为：定义A[**N\*M\*S**] 访问A[(I\***M**+J)\***S**+K]。

其他类型的多维数组依次类推。

3）while实现任意其他循环

i.对于for -> while：

|  |  |
| --- | --- |
| for (<A>;<B>;<C>) {  <D>;  } | <A>;  while ([<B>|true]) {  <D>;  <C>;  } |

ii.对于do .. while -> while：

|  |  |
| --- | --- |
| do {  <A>  } while (<B>); | flag=true;  while (flag || <B>) {  <A>  flag=false;  } |

4）if实现任意其他条件分支

i.对于switch .. case 到 if：

switch (<A>) {

case <N[1]> : <Do[1]>;

case <N[1]> : <Do[2]>;

...

case <N[K]> : <Do[K]>;

default : <DoAny>;

case <N[K+1]> : <Do[K+1]>;

...

case <N[P]> : <Do[P]>;

}

可以等价地表示为：

do {

val=<A>;

nomatch=flag=false;

if (!nomatch) {

if (flag || val==<N[1]>)

{ <Do[1]>; flag=true; }

if (flag || val==<N[2]>)

{ <Do[2]>; flag=true; }

...

if (flag || val==<N[K]>)

{ <Do[K]>; flag=true; }

}

if (flag || nomatch)

{ <DoAny>; flag=true; }

if (flag || val==<N[K+1]>)

{ <Do[K+1]>; flag=true; }

...

if (flag || val==<N[P]>)

{ <Do[P]>; flag=true; }

nomatch=true;

} while (!flag);

这种转换规则具有相等的时间复杂度量级，且是通用的。

4）if消除其他影响控制流的语句（break/continue/return）

i.对于break -> continue：

|  |  |
| --- | --- |
| while (<A>) {  ...开始进入嵌套  break; // 任意层  ...退出所有嵌套  } | flag=true;  while (flag && <A>) {  ...开始进入嵌套  {flag=false; continue;}  ...退出所有嵌套  } |

ii.对于continue -> 有限if：

while (<A>) {

<A[0]>

[if/elseif/else:1] {

<A[1]>

[if/elseif/else:2] {

<A[2]>

[if/elseif/else:3] {

<A[3]>

...

[if/elseif/else:K-1] {

<A[K-1]>

[if/elseif/else:K] {

<C>

continue; // 处于任意K层条件分支内

... // 不可达冗余，可直接消除

}

<B[K-1]>

}

...

<B[3]>

}

<B[2]>

}

<B[1]>

}

<B[0]>

}

可以等价地表示为：

while (<A>) {

flag=true;

<A[0]>

[if/elseif/else:1] {

<A[1]>

[if/elseif/else:2] {

<A[2]>

[if/elseif/else:3] {

<A[3]>

...

[if/elseif/else:K-1] {

<A[K-1]>

[if/elseif/else:K] {

<C>

flag=false;

}

if (flag)

<B[K-1]>

}

...

if (flag)

<B[3]>

}

if (flag)

<B[2]>

}

if (flag)

<B[1]>

}

if (flag)

<B[0]>

}

iii. return与continue类似，通过有限if逐层退出嵌套并返回。

5）利用可消除语句while消除goto语句：

goto语句分为两种，一种为向前，另一种为向后，两种转换方式不同。

i.对于goto(forward) -> do .. while + break

|  |  |
| --- | --- |
| <A>  ...开始进入嵌套  goto L1 // 任意层  ...退出所有嵌套  <B>  L1: <C> | <A>  do {  ...开始进入嵌套  break;  ...退出所有嵌套  <B>  } while (false);  <C> |

ii.对于goto(backward) -> while + continue

|  |  |
| --- | --- |
| <A>  L1: <B>  <C>  ...开始进入嵌套  goto L1 // 任意层  ...退出所有嵌套  <D> | <A>  while (true) {  <B>  <C>  ...开始进入嵌套  continue;  ...退出所有嵌套  <D>  break;  } |

通过上述的若干变换规则，可以抽象为语句归一化的拓扑图结构：



经过了上述的等价性变换，子集的完备性得以证明。

关于宏、指针的等价性说明：

a）宏用于编译前期的代码替换（包含/条件宏等），不影响逻辑表示的等价性；

b）指针在对自定义数组访问时，都可以等价转换为下标访问，其他内存空间访问对ERL自然不可能支持，况且对于算法类计算问题，不存在对指针的这类使用。

* + 1. 活跃全局变量分析：过程对全局变量定值算法的正确性

我们考虑一种最懒惰的方法：无论哪个过程，把任意它可以访问到的全局变量（即全局变量去除被局部变量名所覆盖的其他所有变量）都加到该过程的形参表中，供过程内访问使用，待过程进行返回时，把所有这些新加的全局变量（无论是否重新定值）加入返回值集合。

该证明过程比较简单和显然。

第一：对于过程被调用方，因为过程内定值和使用的变量要么是局部变量，要么是全局变量，懒惰方法把所有全局变量都以形参形式提供给每个过程，因此不会存在过程内变量访问未定义的情况；

第二：对于子过程调用方，调用前与调用后全局变量的值可能发生了变化，所有变化通过子过程的返回值集返回“全部”全局变量的最新值，因此子过程调用之后，任意可能发生值改变的全局变量都会在子过程的返回值中体现并为子过程调用方后续使用。

我们的优化目标是去除其中存在的所有不必要全局变量子集。对于全局变量定值问题而言，如果调用该过程都一定不会对某个全局变量进行定值，那么这个全局变量相对于这个过程就是不必要的。

引理：如果过程P对全局变量g进行了定值，那么该定值要么来自于该过程某个赋值语句(g=)，要么来自于其中任意一个子过程调用，这个子过程直接或间接地对全局变量g进行了定值。

该引理能够很直观地辅助证明全局变量定值算法的正确性。在全局变量定值算法最终迭代完成后，任何全局变量定值都将传播到任何间接对其定值的每个过程。因为同时调用两个子过程意味着两个子过程各自的全局变量定值都属于自己的全局变量定值，所以meet操作是union。

* + 1. 活跃全局变量分析：过程对全局变量使用算法的正确性

该证明同样是基于前述的懒惰方法，即优化目标是去除其中存在的所有不必要全局变量子集。而对于全局变量使用问题而言，如果调用该过程“在入口”一定不会对某个全局变量进行使用，那么这个全局变量相对于这个过程就是不必要的。

引理：如果过程P在入口处对全局变量g进行了使用，那么该使用要么来自于该过程某个非子过程调用语句对其的使用，要么来自于其中任意一个子过程调用，这个子过程直接或间接地在入口处对全局变量g进行了使用，同时两种情况都没有被其之前的全局变量定值所杀死。

同样，迭代过程保证了任何没有被杀死的全局变量的使用被传递到每个过程的入口，并作为子过程调用方的使用，所以全局变量使用也具有可传递的性质。另外，该算法与过程内活跃变量分析具有相同的理论基础，所以meet操作是union。

1. 总结

首先，我们选取了C语法的一个完备子集作为转换算法的输入，并证明了子集语法的完备性，目的是通过完备性说明我们后续的转换算法将会适用于C语法的全集。

其次，我们就两种表示形式介绍了不同的转换算法，分别具有不同的优缺点和适用范围。两种表示形式的差异有以下两个方面：

1）循环是导致指令式语言过程和函数式语言函数不一一对应的最关键原因，对于任意含有循环的指令式语言的一个过程，其函数式语言函数往往需要多个函数联立，换句话说，指令式语言过程和函数式语言函数是一对多的关系。标准函数表示形式借助尾递归转换法转换为直接尾递归，而Erlang函数式语言表示形式借助有限状态机转换为间接尾递归。

2）是否约束单赋值是指令式语言过程和函数式语言函数第二个显著区别。标准函数表示形式借助表达式代入法消除任意可能的赋值，而Erlang函数式语言表示形式借助SSA重命名的方式保持源代码计算拓扑顺序实现单一赋值。

3）标准函数表示形式适用于分支较少的指令式过程，同时除存在公共子表达式之外具有最佳的表示质量（无死代码、自然地常量传播）；而Erlang函数式语言表示形式需要额外进行死代码消除、常量传播等优化，同时输出代码相对较长（但并没有引入额外冗余的计算，运行时间在量级上是相等的）。

此外，我们就如何将全局变量转换为局部变量进行了分析、设计变换算法并证明了其正确性。本质上这也属于优化之一，这是因为该变换算法的主要目的是尽可能减少“不必要的全局变量”传给一个“不需要这些全局变量”的过程，所带来的额外赋值、空间占用等开销。

至于项目目前的不足，有以下两点：

1）主要精力用于实现Erlang函数式语言表示形式的转换，CSC所支持的功能基本都完成了转换的实现，但标准数学函数表示形式的转换尚未处理数组、全局变量等情况；

2）尝试了若干代码行100以内的自实现算法，暂时没有发现BUG，但一般来说的确还有应该很多潜在没发现的BUG，难以短时间内发现。

1. 附实例测试
   1. 标准数学函数实例测试

1）求斐波那契数列第N项

2）判断正整数N是否是素数

(尚不支持全局变量、数组)

以求斐波那契数列第N项为例，输入代码为：

auto f(N) {

if (N<3)

Result=1;

else {

A=1,B=1;

while (N>2) {

A=A+B;

B=A-B;

N=N-1;

}

Result=A;

}

}

输出结果为：

<<function>> funs\_0(A, B, N)

--> funs\_0 ( A + B , A , N - 1 ) ; When ( N > 2 )

--> { A , B , N } ; When ( ( N > 2 ) == false )

<<function>> f(N)

--> 1 ; When ( N < 3 )

--> element ( 1 , funs\_0 ( 1 , 1 , N ) ) ; When ( ( N < 3 ) == false )

其他样例转换详见conv-math中的相关源码。

* 1. Erlang函数式语言实例测试

1）求斐波那契数列第N项

2）判断正整数N是否是素数

3）递归实现的冒泡排序

4）分治快速排序

以分支快速排序为例，输入代码为：

const auto MAX=100000;

auto arr[MAX];

void generate() {

auto i;

i=0;

while (i<MAX) {

arr[i]=100\*i\*i%31;

i=i+1;

}

}

void qsort(auto l,auto r) {

auto i, j, flag, tmp;

i=l;

j=r;

tmp=arr[i];

while (i<j) {

flag=0;

if (i<j) {

if (tmp<=arr[j]) {

flag=1;

}

}

while (flag>0) {

j=j-1;

flag=0;

if (i<j) {

if (tmp<=arr[j]) {

flag=1;

}

}

}

arr[i]=arr[j];

flag=0;

if (i<j) {

if (tmp>=arr[i]) {

flag=1;

}

}

while (flag>0) {

i=i+1;

flag=0;

if (i<j) {

if (tmp>=arr[i]) {

flag=1;

}

}

}

arr[j]=arr[i];

}

arr[i]=tmp;

if (l<i-1) {

qsort(l, i-1);

}

if (r>i+1) {

qsort(i+1, r);

}

}

void output() {

auto i;

i=0;

while (i<MAX) {

print(arr[i]);

println();

i=i+1;

}

}

void main() {

generate();

qsort(0, MAX-1);

output();

}

输出结果为：

-module(qsort).

-export([entry\_main/0, generate/1, qsort/3, output/1, main/1]).

generate(\_arr) ->

generate\_b1(\_arr).

generate\_b1(\_arr) ->

Id1\_i = 0,

generate\_b2(Id1\_i, \_arr).

generate\_b2(Id2\_i, Id1\_arr) ->

case (Id2\_i < 100000) of

false -> generate\_b4(Id1\_arr);

true -> generate\_b3(Id2\_i, Id1\_arr)

end.

generate\_b3(Id2\_i, Id1\_arr) ->

T2 = 100 \* Id2\_i,

T3 = T2 \* Id2\_i,

T4 = T3 rem 31,

Id2\_arr = setelement(1 + Id2\_i, Id1\_arr, T4),

T6 = Id2\_i + 1,

Id3\_i = T6,

generate\_b2(Id3\_i, Id2\_arr).

generate\_b4(\_arr) ->

{\_arr}.

qsort(\_r, \_l, \_arr) ->

qsort\_b1(\_l, \_r, \_arr).

qsort\_b1(Id0\_l, Id0\_r, Id0\_arr) ->

Id1\_i = Id0\_l,

Id1\_j = Id0\_r,

T2 = element(1 + Id1\_i, Id0\_arr),

Id1\_tmp = T2,

qsort\_b2(Id1\_i, Id0\_l, Id1\_j, Id0\_r, Id0\_arr, Id1\_tmp).

qsort\_b2(Id1\_i, \_l, Id1\_j, \_r, Id1\_arr, \_tmp) ->

case (Id1\_i < Id1\_j) of

false -> qsort\_b20(Id1\_i, \_l, \_r, Id1\_arr, \_tmp);

true -> qsort\_b3(Id1\_i, \_l, Id1\_j, \_r, Id1\_arr, \_tmp)

end.

qsort\_b3(Id1\_i, \_l, Id1\_j, \_r, \_arr, \_tmp) ->

Id2\_flag = 0,

case (Id1\_i < Id1\_j) of

false -> qsort\_b6(Id1\_i, \_l, Id1\_j, \_r, \_arr, \_tmp, Id2\_flag);

true -> qsort\_b4(Id1\_i, \_l, Id1\_j, \_r, \_arr, \_tmp, Id2\_flag)

end.

qsort\_b20(Id1\_i, Id0\_l, \_r, Id1\_arr, Id1\_tmp) ->

Id4\_arr = setelement(1 + Id1\_i, Id1\_arr, Id1\_tmp),

T39 = Id1\_i - 1,

case (Id0\_l < T39) of

false -> qsort\_b22(Id1\_i, \_r, Id4\_arr);

true -> qsort\_b21(Id1\_i, Id0\_l, \_r, Id4\_arr)

end.

qsort\_b4(\_i, \_l, Id1\_j, \_r, Id1\_arr, Id1\_tmp, \_flag) ->

T7 = element(1 + Id1\_j, Id1\_arr),

case (Id1\_tmp > T7) of

true -> qsort\_b6(\_i, \_l, Id1\_j, \_r, Id1\_arr, Id1\_tmp, \_flag);

false -> qsort\_b5(\_i, \_l, Id1\_j, \_r, Id1\_arr, Id1\_tmp)

end.

qsort\_b6(\_i, \_l, \_j, \_r, \_arr, \_tmp, Id3\_flag) ->

case (Id3\_flag > 0) of

false -> qsort\_b11(\_i, \_l, \_j, \_r, \_arr, \_tmp);

true -> qsort\_b7(\_i, \_l, \_j, \_r, \_arr, \_tmp)

end.

qsort\_b21(Id1\_i, Id0\_l, \_r, Id4\_arr) ->

T41 = Id1\_i - 1,

{Id5\_arr} = qsort(T41, Id0\_l, Id4\_arr),

qsort\_b22(Id1\_i, \_r, Id5\_arr).

qsort\_b22(Id1\_i, Id0\_r, Id6\_arr) ->

T43 = Id1\_i + 1,

case (Id0\_r > T43) of

false -> qsort\_b24(Id6\_arr);

true -> qsort\_b23(Id1\_i, Id0\_r, Id6\_arr)

end.

qsort\_b5(\_i, \_l, \_j, \_r, \_arr, \_tmp) ->

Id3\_flag = 1,

qsort\_b6(\_i, \_l, \_j, \_r, \_arr, \_tmp, Id3\_flag).

qsort\_b7(Id1\_i, \_l, Id1\_j, \_r, \_arr, \_tmp) ->

T11 = Id1\_j - 1,

Id2\_j = T11,

Id4\_flag = 0,

case (Id1\_i < Id2\_j) of

false -> qsort\_b10(Id1\_i, \_l, Id2\_j, \_r, \_arr, \_tmp, Id4\_flag);

true -> qsort\_b8(Id1\_i, \_l, Id2\_j, \_r, \_arr, \_tmp, Id4\_flag)

end.

qsort\_b11(Id1\_i, \_l, Id1\_j, \_r, Id1\_arr, \_tmp) ->

T19 = element(1 + Id1\_j, Id1\_arr),

Id2\_arr = setelement(1 + Id1\_i, Id1\_arr, T19),

Id6\_flag = 0,

case (Id1\_i < Id1\_j) of

false -> qsort\_b14(Id1\_i, \_l, Id1\_j, \_r, Id2\_arr, \_tmp, Id6\_flag);

true -> qsort\_b12(Id1\_i, \_l, Id1\_j, \_r, Id2\_arr, \_tmp, Id6\_flag)

end.

qsort\_b23(Id1\_i, Id0\_r, Id6\_arr) ->

T45 = Id1\_i + 1,

{Id7\_arr} = qsort(Id0\_r, T45, Id6\_arr),

qsort\_b24(Id7\_arr).

qsort\_b24(Id8\_arr) ->

{Id8\_arr}.

qsort\_b8(\_i, \_l, Id2\_j, \_r, Id1\_arr, Id1\_tmp, \_flag) ->

T15 = element(1 + Id2\_j, Id1\_arr),

case (Id1\_tmp > T15) of

true -> qsort\_b10(\_i, \_l, Id2\_j, \_r, Id1\_arr, Id1\_tmp, \_flag);

false -> qsort\_b9(\_i, \_l, Id2\_j, \_r, Id1\_arr, Id1\_tmp)

end.

qsort\_b10(\_i, \_l, \_j, \_r, \_arr, \_tmp, \_flag) ->

qsort\_b6(\_i, \_l, \_j, \_r, \_arr, \_tmp, \_flag).

qsort\_b12(Id1\_i, \_l, \_j, \_r, Id2\_arr, Id1\_tmp, \_flag) ->

T23 = element(1 + Id1\_i, Id2\_arr),

case (Id1\_tmp < T23) of

true -> qsort\_b14(Id1\_i, \_l, \_j, \_r, Id2\_arr, Id1\_tmp, \_flag);

false -> qsort\_b13(Id1\_i, \_l, \_j, \_r, Id2\_arr, Id1\_tmp)

end.

qsort\_b14(\_i, \_l, \_j, \_r, \_arr, \_tmp, Id7\_flag) ->

case (Id7\_flag > 0) of

false -> qsort\_b19(\_i, \_l, \_j, \_r, \_arr, \_tmp);

true -> qsort\_b15(\_i, \_l, \_j, \_r, \_arr, \_tmp)

end.

qsort\_b9(\_i, \_l, \_j, \_r, \_arr, \_tmp) ->

Id5\_flag = 1,

qsort\_b10(\_i, \_l, \_j, \_r, \_arr, \_tmp, Id5\_flag).

qsort\_b13(\_i, \_l, \_j, \_r, \_arr, \_tmp) ->

Id7\_flag = 1,

qsort\_b14(\_i, \_l, \_j, \_r, \_arr, \_tmp, Id7\_flag).

qsort\_b15(Id1\_i, \_l, Id1\_j, \_r, \_arr, \_tmp) ->

T27 = Id1\_i + 1,

Id2\_i = T27,

Id8\_flag = 0,

case (Id2\_i < Id1\_j) of

false -> qsort\_b18(Id2\_i, \_l, Id1\_j, \_r, \_arr, \_tmp, Id8\_flag);

true -> qsort\_b16(Id2\_i, \_l, Id1\_j, \_r, \_arr, \_tmp, Id8\_flag)

end.

qsort\_b19(Id1\_i, \_l, Id1\_j, \_r, Id2\_arr, \_tmp) ->

T35 = element(1 + Id1\_i, Id2\_arr),

Id3\_arr = setelement(1 + Id1\_j, Id2\_arr, T35),

qsort\_b2(Id1\_i, \_l, Id1\_j, \_r, Id3\_arr, \_tmp).

qsort\_b16(Id2\_i, \_l, \_j, \_r, Id2\_arr, Id1\_tmp, \_flag) ->

T31 = element(1 + Id2\_i, Id2\_arr),

case (Id1\_tmp < T31) of

true -> qsort\_b18(Id2\_i, \_l, \_j, \_r, Id2\_arr, Id1\_tmp, \_flag);

false -> qsort\_b17(Id2\_i, \_l, \_j, \_r, Id2\_arr, Id1\_tmp)

end.

qsort\_b18(\_i, \_l, \_j, \_r, \_arr, \_tmp, \_flag) ->

qsort\_b14(\_i, \_l, \_j, \_r, \_arr, \_tmp, \_flag).

qsort\_b17(\_i, \_l, \_j, \_r, \_arr, \_tmp) ->

Id9\_flag = 1,

qsort\_b18(\_i, \_l, \_j, \_r, \_arr, \_tmp, Id9\_flag).

output(\_arr) ->

output\_b1(\_arr).

output\_b1(\_arr) ->

Id1\_i = 0,

output\_b2(Id1\_i, \_arr).

output\_b2(Id2\_i, \_arr) ->

case (Id2\_i < 100000) of

false -> output\_b4();

true -> output\_b3(Id2\_i, \_arr)

end.

output\_b3(Id2\_i, Id0\_arr) ->

T2 = element(1 + Id2\_i, Id0\_arr),

io:put\_chars(integer\_to\_list(T2)),

io:put\_chars([10]),

T5 = Id2\_i + 1,

Id3\_i = T5,

output\_b2(Id3\_i, Id0\_arr).

output\_b4() ->

ok.

main(\_arr) ->

main\_b1(\_arr).

main\_b1(Id0\_arr) ->

{Id1\_arr} = generate(Id0\_arr),

T1 = 100000 - 1,

{Id2\_arr} = qsort(T1, 0, Id1\_arr),

output(Id2\_arr),

{Id2\_arr}.

entry\_main() ->

\_arr = erlang:make\_tuple(100000, 0),

main(\_arr),

ok.

其他样例转换详见conv-erlang中的相关源码。