分支前提编程模型

系统定义

类型

基础数据类型只有一种，word，定长存储空间。由于只有一种数据类型，后面不再提及任何数据类型。值得注意的是，word是定长的，因此取值范围有限。

指令集

η 为该模型的指令机，$\eta(i_1,i_2,\dots,i_n,op)[r]\{\emptyset\}$意思是，该机器接受 $i_1,i_2,\dots,i_n$ 作为参数，执行编号为 op 的指令，得到结果 r 。后面的大括号指示代码块集合，由于 η 是最基本的指令，故代码块集合为空。

值得注意的是， 由于 η 格式的通用性，假设其所有指令输入参数个数都一样， op 只要存在都合法；返回值仅有一个。

结构

基础结构 Code Block/Pure Function

代码块，仅包含若干基础指令的一块代码。是所有结构的最基础部分。形式描述为 $\sigma(i_1,\dots,i_m)[r_1,\dots,r_n]\{\eta_1,\dots,\eta_n\}$。参考指令集的描述方法，代码块由若干指令组成。输入为 $i_1,\dots,i_m$ 输出为 $r_1,\dots,r_n$ 。其中 $\eta_k(i_{k_1},\dots,i_{k_s})[r_k]\{\emptyset\}$ 。

值得注意的是 $\sigma()$ 到 $\eta_k()$ 的转换，我们假设当所有参数符合要求的时候，每个参数都能"去它应该去的位置"，就不再向下讨论更具体的实现了。

分支 Branch

分支是一种对数据影响代码块执行的描述，形式化描述为 $\mu(i_1,\dots,i_n)[r_1,\dots,r_m]\{f_0,\{f_{11},\dots,f_{1m}\},\dots,\{f_{t1},\dots,f_{tm}\}\}$ 。其中在代码块中为t个代码块的集合，另加一个f_0 。分支执行逻辑为：首先由 $f_0(i_{n1},\dots,i_{ns})[s]\{_\}$计算出代码块索引 s ，由 s 得知选择 $\{f_{s1},\dots,f_{sm}\}$作为接下来继续进行计算的代码块。

循环 Loop

循环是一种同一代码块不断执行的结构，形式化描述为 $\upsilon(itr,end,i_1,\dots,i_n)[r_1,\dots,r_m]\{f_{01},f_{02},\dots,f_m\}$其中值得注意的是$itr$ ，它指示当前迭代次数；以及 end 它指示循环总次数。

连接 Joint

两个代码块连接，记作f_1+f_2 。

有两个代码块 $f_1(i_1,\dots,i_a)[r_1,\dots,r_b]\{_\}$与 $f_2(i_1,\dots,i_c)[r_1,\dots,r_d]\{\_\}$，其满足要求 $f_1[]\subset f_2()$ ，即f_1 的输出能够满足f_2的输入，那么形成新的代码块$f'(i_1,\dots,i_a)[r_1,\dots,r_d]\{_\}$。

其中最值得注意的是省略的那部分$\{_\}$ 。假设$f_1\{f_{11},\dots,f_{1m}\}$ $f_2\{f_{21},\dots,f_{2n}\}$，设 $f_1+f_2\{f_{31},\dots,f_{3n}\}$， 那么对于 f_{3k} 相当于 $f_{2k}(f_1[])$ 。

值得注意的是，连接不是嵌套，不适用于分支与循环的嵌套。

函数 Function

函数是一切有输入输出代码块的统称，形式化描述为 $f(i_1,\dots,i_n)[r_1,\dots,r_m]\{f_1,\dots,f_m\}$。其形式与基础结构几乎完全一致，区别在于后面的代码块集合为函数集合。

函数的构成：

$$f\to\eta\mid\sigma\mid\mu\mid\upsilon\mid f_1+f_2+\dots+f_n$$

用自然语言描述为，函数的内部可能由函数的嵌套、连接构成。最基础的函数类型为基础代码块、分支和循环。

分支前提 Branch First

函数内部分支前提

设

$$f \equiv f_1 + \ldots + f_{i-1} + \mu\{f_{i_0}, f_{i_1}, \ldots, f_{i_m}\} + f_{i+1} + \ldots + f_n$$

前提

$$f' \equiv \mu\{f_{i_0}(f_{i-1}(f_{i-2}(\ldots(f[])[])\ldots)[]), \{f_1 + \ldots + f_{i_1} + \ldots + f_n\}, \{f_1 + \ldots + f_{i_2} + \ldots + f_n\}, \ldots, \{f_1 + \ldots + f_{i_m} + \ldots + f_n\}\}$$

嵌套分支合并

首先拓展分支的形式：

$$M\{\{f_{01}, f_{02}, \ldots, f_{0n}\}, \{\ldots\{f_{1\ldots1}, \ldots, f_{1\ldots n_1}\}\ldots\}, \ldots\}$$

首先由输入，计算出

$$\{s_1, \ldots, s_n\}$$

其中 $f_{0k}[s_k]$ 。由 s_k 选择第 $\{f_{\ldots s_k \ldots}, \ldots\}$

的代码块集合，直到 s_n 选择出唯一的代码块，作为拓展分支选择执行的块。

设

$$\mu_1\{f_0, f_1, \ldots, f_{i-1}, \mu_i\{f_{i_0}, f_{i_1}, \ldots, f_{i_m}\}, f_{i+1}, \ldots, f_n\}$$

合并

$$M\{f_0, f_{i0}(f_{i-1}(\ldots f_1(\mu_1())[]\ldots)[])\}$$

分支外提

设

$$f\{f_1, \ldots, \mu_i\{f_{i_0}, f_{i_1}, \ldots, f_{i_n}\}, \ldots, f_m\}$$

分支外提：

$$\mu f_{i_0}(f_{i-1}(\ldots f_1(f())[]\ldots)[]), \{f_1, \ldots, f_{i_1}, \ldots, f_m\}, \ldots, \{f_1, \ldots, f_{i_n}, \ldots, f_m\}$$

最终

理论上，所有函数内的分支可以提到函数最前，并加以合并，最后外提；迭代处理后，最终实现全部分支前提。

表达能力

直叙（无特殊结构）

首先需要确定 ISA，定义 η 。通过函数的嵌套实现所有的计算。

反复（循环）

下面以 C 语言的 while 为例讨论这个问题。

在循环外提之前，while 语句可以写作 $\upsilon(itr, end, i_1, \ldots)[itr', end', r_1, \ldots]\{f_{01}, f_{02}.f_1, \ldots\}$，对于$end$由谁决定、是否可变，分为两种情况。

定值循环

end 为定值，itr 的变化只与机器实现有关，不能作为输出，但可以作为输入，但不能作为分支判断的输入。

变值循环

end 与 itr 的值由代码块更新。变值循环没有办法使用前面的分支前提方法，因此需要将变值循环改写为定值循环。

综上所说，由于分支前提的限制。在可以使用上述分支前提的方法限制内，只可以写定值循环，其循环次数在循环开始前已经决定，或者为定值编入程序。而且不能使用与迭代子相关的数据作为判断。

总分（分支）

若将 $\mu\{f_0,f_1,f_2\}$ 看做 if 语句，则 f_0 为分支条件判定，f_1 、 f_2为 $true$ 与 $false$ 时的分支。

联想（递归）

由上面的分支前提方法得知，任何形式的不定长度递归，如$f(f(\dots f()[]\dots)[])[]$或 $f_1(f_2(\dots f_1(\dots f_2(\dots)[]\dots)[]\dots)[])$都会在分支外提时产生死循环。因为分支前提看来，递归是有代码块无穷嵌套的“可能性”，就会试图产生这样的选择代码块。

完备性

由于上述限制，这个模型是否图灵完备，其实我不太清楚。

外延

分支前提后，代码块的合并。对于选中所有前提的分支，在输入一定时，代码块的选择就已经决定好。可以考虑使用 $+$ 对所有代码块$\{f_1,f_2,\dots,f_m\}$可合并为 $f_1+f_2+\dots+f_m$ 。对于若 $\eta\langle+,*\rangle$ 的线性模型，运算可由矩阵表示的线性空间计算，设 $\{f_1,f_2,\dots,f_m\}$的计算矩阵表示为 $\{F_1,F_2,\dots,F_m\}$，则 $f_1+f_2+\dots+f_m$ 表示为 $F_1\times F_2\times\dots\times F_m$ 得到一个矩阵。

问题

表达能力限制

比如不支持全面的循环，不支持函数的任意形式递归等可能导致死循环或死递归的表达。

I/O

由这个模型，得知所有运算均在一个封闭空间内进行。并不涉及任何形式的输入输出。可以考虑只将代码某一块进行分支前提处理，而在其他部分进行 I/O等无法再这个模型中表达的操作。

硬件实现

硬件实现可能和大多数硬件不一样，仅能实现模拟器。

庞大分支

如果在编译时实现所有的分支打表处理，那么对于可能由多层嵌套产生的分支空间是非常庞大的。

适用范围

分支前提的目的主要是为了将分支计算和数值计算分开。对于某些“讨厌”分支的场合，可以提前先将所有分支算好，再直接由得到的代码放到那些讨厌分支的硬件上运行。其它适用范围或者模型的改进可能还需要后续发展。