约束计算复杂度的类型系统

史杨勋惟 1200012741 信息科学技术学院
蔡思培 1100011788 信息科学技术学院

2015,06,16

1 简介

程序的复杂度探索已经成为了 21 世纪以来计算机科学领域最为炙手可热的话题之一。从编程语言的类型系统设计上，这个问题也已经得到了不少深入的研究。这些研究中既包括了显式地将计算复杂度嵌入类型系统，也包括隐式得通过类型系统来约束复杂度。

这次大作业我们针对命令式编程语言设计了一个能够约束计算复杂度的类型系统。在这个系统中，所有 Well-Type 的且能够终止的程序都能在多项式时间内计算，这也就意味着在这个类型系统上的所有 Well-Type 程序如果能够终止，那么计算的复杂度一定是多项式的。

本类型系统不支持高阶方法和递归嵌套，但对命令式编程中最基本的循环和分支语句均有良好的支持。为了证明这个类型系统的实用性，这篇报告中会有几个示例程序，更进一步，还有证明所有在确定性图灵机上多项式时间内可解的问题均可以用一个 Well-Type 的程序模拟。

在我们代码的实现中，我们完成了类型检查的所有部分。Evaluation 部分由于时间原因没有完成，但是由于这个代码和传统的命令式编程如 C 语言是兼容的，所以 Evaluation 部分也不是重点。

2 核心思想

对于一个没有递归和高阶方法的程序语言来说，复杂度的约束主要实现在循环的控制上。如果一个变量出现在循环的判断体中，那么这个变量就会对计算复杂度产生直接的影响。与此同时，如果另一个变量在程序中有数据流流入一个会对复杂度产生直接影响的变量，那么这个变量就会间接地对程序的复杂度产生影响。我们的类型系统核心的设计思想是通过类型来描述这种数据流之间的关系。

一种最直觉的想法就是将所有直接或者间接对复杂度产生影响的变量归为同一类型，不产生影响的变量归为另一类型，但是在具体设计的过程中我们发现在引入运算符和表达式的类型之后，仅仅通过一个维度来划分是不够的，所以我们将类型扩展为一个二元组 (a, b) $a, b \in \{0, 1\}$

3 类型系统

3.1 Term

$$
\begin{aligned}
E &::= X \mid op(E_1, ..., E_n) \\
C &::= X := E \\
&\quad \mid while\ (E)\ C \\
&\quad \mid if\ E\ then\ C\ else\ C' \\
&\quad \mid C\ ;\ C' \\
T &::= (a, b)\ a, b \in \{0, 1\}
\end{aligned}
$$

以上是一个最基本的命令式编程语言的 Term，其中包含了 while 循环和 if 分支语句。

3.2 Typing Rules

$$\frac{\Gamma(X) = a}{\Gamma, \Delta \vdash X : (a, b)\ where\ b \leqslant a} \quad T\_Var$$

$$\frac{\Gamma, \Delta \vdash E_1 : (a_1, b_1), \cdots, E_n : (a_n, b_n)}{\Gamma, \Delta \vdash op(E_1, \cdots, E_n) : (a, b)\ \ where (a_1, b_1) \to \cdots \to (a_n, b_n) \to (a, b) \in \Delta(op)} \quad T\_Op$$

$$\frac{\Gamma(X) = a',\ \Gamma, \Delta \vdash E : (a, b)\ and\ a' \leqslant a}{\Gamma, \Delta \vdash X := E : (a, b)} \quad T\_Assign$$

$$\frac{\Gamma, \Delta \vdash C : (a, b), C' : (a', b')}{\Gamma, \Delta \vdash C; C' : (a \vee a', b \vee b')} \quad T\_Seq$$

$$\frac{\Gamma, \Delta \vdash E : (a, a'), C : (a, b)\ where\ b < a}{\Gamma, \Delta \vdash while(E)C : (a, b)} \quad T\_While$$

$$\frac{\Gamma, \Delta \vdash E : (\rho, \rho'), C : (a, b), C' : (a, b)\ where\ a \leqslant \rho}{\Gamma, \Delta \vdash if\ E\ then\ C\ else\ C' : (a, b)} \quad T\_If$$

其中 Γ 的初始值为程序员指定的初始变量及其初始类型（一共只有两种类型）的集合，Δ 的初始值为程序员指定的每个在程序中出现的运算符及其类型的集合（一共也只有两种类型，后面会详述）。

3.3 Semantics With Timing

对于命令式编程语言来说，语义其实就是描述状态变化的规则，这一点和我们上课所学的 Reference 比较类似。这里采用了 big-step 的规则，这样做的好处是可以把每次推导的时间进行统计。以下是规则：

$$\frac{}{\mu \models X \Rightarrow^1 \mu(X)} \quad E\_Var$$

$$\frac{\mu \models E_1 \Rightarrow^{t_1} d_1, \cdots, E_n \Rightarrow^{t_n} d_n}{\mu \models op(E_1 \cdots E_n) \Rightarrow^{\Sigma_{i=1}^{n} t_i} [op](d_1 \cdots d_n)} \quad E\_Op$$

$$\frac{\mu \models E \Rightarrow^{t} d}{\mu \models X := E \Rightarrow^{t+1} \mu[X \leftarrow d]} \quad E\_Assign$$

$$\frac{\mu \models E \Rightarrow^{t} \mu' \quad \mu' \models E \Rightarrow^{t'} \mu''}{\mu \models C; C' \Rightarrow^{t+t'} \mu''} \quad E\_Seq$$

$$\frac{\mu \models E \Rightarrow^{t} true, C \Rightarrow^{t'} \mu'}{\mu \models if\ E\ then\ C\ else\ C' \Rightarrow^{t+t'+1} \mu'} \quad E\_Ift$$

$$\frac{\mu \models E \Rightarrow^{t} false, C' \Rightarrow^{t'} \mu'}{\mu \models if\ E\ then\ C\ else\ C' \Rightarrow^{t+t'+1} \mu'} \quad E\_Iff$$

$$\frac{\mu \models E \Rightarrow^{t} false}{\mu \models while(E)C \Rightarrow^{t} \mu} \quad E\_Whilef$$

$$\frac{\mu \models E \Rightarrow^{t} true, C \Rightarrow^{t'} \mu' \quad \mu' \models while(E)C \Rightarrow^{t''} \mu''}{\mu \models while(E)C \Rightarrow^{t+t'+t''+1} \mu''} \quad E\_Whilet$$

3.3.1 复杂度约束简述

这里我们考虑的复杂度是基于 Word 的。设 d 是一个 word，记 $|d|$ 为 d 的长度。如果 $\mu_0[\overrightarrow{X} \leftarrow \overrightarrow{d}] \models C \Rightarrow^{t} \mu$，则 C 在 $\overrightarrow{d}$ 上的执行时间是 t。如果对于任意 $\overrightarrow{d}$ 都存在一个多项式 $Q(x)$ 使得 $t \leqslant Q(max(|d_i|)$，那么这个指令就是多项式时间可计算的。

3.4 Operators

显然，不同的操作对复杂度的影响程度是不一样的。比如一个变量可能影响复杂度，那么对这个变量的加法就可以认为是一个危险的运算，而减法可以认为是一个安全的运算。这里我们规定了两种基于 word 的运算，分别称为安全运算和半安全运算。他们的定义和类型约定如下：

安全运算：一个运算被称之为安全运算当它（1）是一个判断运算（返回 true 或者 false），或者（2）对于所有的操作数 $d_1, \cdots, d_n$，存在 i 使得 $[op](d_1, \cdots, d_n)$ 是 d_i 的子串。在正整数上的 -1 运算就是安全运算（把自然数 n 用 n 个 1 的 word 表示）

半安全运算：一个运算被称之为半安全运算当存在一个常数 c 使得 $|[op](d_1, \cdots, d_n)| \leqslant max(d_i) + 1$。例如正整数上的 $+1$ 运算就属于一个半安全运算。

一个安全运算的类型为 $(a_1, b_1) \cdots \to (a_n, b_n) \to (\wedge_{i=1,n} a_i, \vee_{i=1,n} b_i)$

一个半安全运算的类型为 $(a_1, b_1) \cdots \to (a_n, b_n) \to (\wedge_{i=1,n} a_i, \wedge_{i=1,n} a_i)$

如果在一个环境下的所有运算要么是安全的，要么是半安全的，那么我们成为整个环境是一个安全环境，否则为一个不安全环境。例如有乘法运算的环境就是一个不安全环境，因为乘法不是半安全的，更不是安全的接下来，我们所有的论证均在安全环境下推理，即所有的运算符都是安全的。

4 示例程序

接下来我们展示一下用我们的类型系统可以实现的几个基本运算，同时展示类型推导的过程。

4.1 Addition

```
X:1
Y:0
>0,-1:neutral
+1:positive

Add(X,Y->Y):
while(X > 0) {
        X:=X-1;
        Y:=Y+1;
}
```

上面是一个简单的加法的实现，对于上述代码，Type-Derivation 的过程如下

$$\dfrac{\dfrac{\Gamma(X)=1}{\Gamma,\Delta\vdash X>0:(1,1)} \quad \dfrac{\dfrac{\dfrac{\Gamma(X)=1}{\Gamma,\Delta\vdash X:(1,0)}}{\Gamma,\Delta\vdash X-1:(1,0)}}{\Gamma,\Delta\vdash X:=X-1:(1,0)} \quad \dfrac{\dfrac{\dfrac{\Gamma(Y)=0}{\Gamma,\Delta\vdash Y:(0,0)}}{\Gamma,\Delta\vdash Y+1:(0,0)}}{\Gamma,\Delta\vdash Y:=Y+1:(0,0)}}{\Gamma,\Delta\vdash X:=X-1;Y:=Y+1:(1,0)}$$

$$\Gamma,\Delta\vdash while...:(1,0)$$

4.2 Multiplication

下面是乘法的一个实现，Type-Derivation 直接在程序中表述了。

```
X:1
Y:1
Z:0
U:1
>0,-1:neutral
+1:positive

Mul(X,Y,Z->Z: )
Z:=0; :(0,0)
while (X > 0) {
        X:=X-1 :(1,0)
        U:=Y :(1,0)
        while (Y > 0) {
                Y:=Y-1 :(1,0)
                Z:=Z+1 :(0,0)
        } :(1,0)
        Y:=U; :(1,0)
} :(1,0)
```

4.3 Linear Search

下面这个问题是搜索在给定的 01 串中是否存在子串 v。

```
X:1
Y:1
Z:1
tt,ff,==v,pred,==0:neutral

SearchV(X,Y,Z->Z: )

Y:=tt; :(1,0);
Z:=ff; :(1,0);
while(Z) {
        if (X==v) :(1,0)
        {
                then {Y:=ff; Z:=tt} :(1,0)
        }
        else
        {
                if (X==0) :(1,0)
                {
                        then {Y:=ff} :(1,0)
                }
                else X:=pred(X) :(1,0)
        } :(1,0)
} :(1,0)
```

4.4 多项式时间内可解的图灵机问题

下面进一步证明该类型系统的实用性，即每一个在图灵机上多项式时间内可解的问题均可用一个 Well-Type 的程序来模拟。假设复杂度不超过 $O(n^k)$。每一步转移可用下程序来模拟

```
Transition:
if (Right==0)
        then if (State==s0)
                 then {State:=t0;Shift;}
                 else if ...
        else if (State==s1)
                ...
```

上述所有变量类型均为 0, 所以 Transition 的类型为 (0,0)。整个迭代过程可以如下实现.

```
while(X1>0) {
        X1:=X1-1; :(1,0)
        X2:=X; :(1,0)
        while(X2>0) {
                ...
                while(Xk>0) {
                        Xk=Xk-1;          :(1,0)
                        Transition; :(0,0)
                } :(1,0)
        } :(1,0)
} :(1,0)
```

由于上述迭代次数至少有 n^k 次，所以上述程序一定能够终止。与此同时，在类型的推导过程中我们发现它也是 Well-Type 的。

5 性质证明

5.1 Safety

在这个类型系统上，计算的过程是通过状态的不断转变来描述的，也即 $\mu_0 \Rightarrow \mu_1 \Rightarrow \cdots$，而不是传统的 terms 上的 evaluation。当程序 P 的整个指令 C 确定后，它的类型就不会变，也即具有 Preservation，同时，由于整个 term 是不进行归约的，所以程序也是不会因为 term 而 Stuck 的。所以这个类型系统具有 Type Safety。

然而整个过程并不一定会终止，一个 Well-Type 的程序也是有可能无法停机的。但是回到最初的结论，我们的结论是如果一个程序 Well-Type 的程序而且能够终止，那么这个程序的计算时间是多项式级别的。所以程序终止是一个基于的假设，而不是结论。

5.2 复杂度约束的证明

整个证明过程的大致思路是，首先证明每个 Well-Type 的程序中，1 型变量的值是独立于 0 型变量的，其次证明整个程序的循环总次数是由 1 型变量的状态决定的，进而证明 1 型变量的状态具有单调递减性，从而给出一个复杂度的多项式约束。接下来逐步给出整个证明过程的推理步骤，其中大多证明都用到了 Induction，这个和课上讲的诸多证明非常相似，所以有些引理的证明并不会详细展开。另外由于详细证明将会大大增加篇幅，这里的证明大多将通过简单描述的方法给出。

5.2.1 类型的单调性

引理 1 *对任何一个 Well-Type 的程序中的每一个表达式 $E : (a, b)$，均有 $b \leqslant a$。*

证明 1 *对 E 的结构进行归纳即可。*

引理 2 *对任何一个 Well-Type 的程序中的每一个表达式 $E : (a, b)$，对于 E 中出现的每个变量 $X : (a', b')$ 均有 $a' \geqslant a$。*

证明 2 *对 E 的结构进行归纳即可。*

引理 3 *对任何一个 Well-Type 的程序中的每一个表达式 $C : (a, b)$，均有 $b \leqslant a$。*

证明 3 *在 Typing-Derivation-Tree 上进行归纳即可。*

引理 4 *对任何一个 Well-Type 的程序中的每一个表达式 $C : (a, b)$，对于任意 C 中的子命令 $C' : (a', b')$，均有 $a' \leqslant a, b' \leqslant b$。*

证明 4 *在 Typing-Derivation-Tree 上进行归纳即可。*

5.2.2 类型的独立性

引理 5 对任何一个 *Well-Type* 的且能终止的程序，如果 *1* 型变量的初值给定，无论 *0* 型变量如何给定初值，*1* 型变量再每步推导过程中的值都不会变。

证明 5 在 *Typing-Derivation-Tree* 上进行归纳，考虑最后一步 *Typing*。

考虑任何一个出现在赋值语句左边的 *1* 型变量，由 *Typing Rules*，赋值语句右边的表达式也是 *1* 型的。由上一条引理，赋值语句右边的表达式中的所有变量都是 *1* 型的。再由归纳假设，右边的所有变量与 *0* 型变量无关，所以左边的 *1* 型变量的新值也与 *0* 型变量无关。

5.2.3 循环长度约束

引理 6 对任何一个 *Well-Type* 且能够终止的程序，循环的执行次数只和 *1* 型变量的初始状态有关。

证明 6 之前证明了 *1* 型变量在每时每刻的状态只与 *1* 型变量的初始状态有关，于 *0* 型变量无关。而在循环语句的判断式中的变量均为 *1* 型变量，所以从直觉上来看，循环语句的执行次数只和 *1* 型变量的初始状态有关。

这一步的系统化证明需要对循环执行次数设计 *Derivation-Tree*，并且在此树上递归，这里就不展开了。

引理 7 约定一个命令的推导过程是：$\mu_0 \Rightarrow \mu_1 \cdots \Rightarrow \mu_n$，每一步推导经过一次循环。如果 *1* 型变量在推导过程中出现两个关于 *1* 型变量完全相同的状态，那么程序就不会终止。

证明 7 利用反证法，如果程序能够终止如果存在两个相同的 *1* 型变量状态，那么这两个状态推到终止状态经过的循环次数应该相同。而这两个状态又出现在了推导过程的不同位置，所以他们到终态的循环次数不同，矛盾。

5.2.4 安全 Term 的递减性

如果一个 Term 中只出现了安全运算，那么这个 Term 就是一个安全 Term。

引理 8 对任何一个 *Well-Type* 的程序中的表达式 $E : (a, b)$，如果 $a > b$，那么这个表达式就是一个安全 *Term*。

证明 8 对 E 的结构进行归纳即可。

引理 9 对任何一个 *Well-Type* 的程序的指令 C，如果 $C : (1, 0)$，那么在推导过程中 *1* 型变量可能的状态数是多项式级别的。

证明 9 由于所有 *1* 型变量的赋值语句均为安全 *Term*，由安全运算的定义我们知道安全 *Term* 一定是某一个输入 *word* 的子串。由于输入的 *word* 个数有限，而不同子串个数有小于长度的平方，所以每一个 *1* 型变量可能被赋予的值是在多项式个数内的，进而所有 *1* 型变量的可能状态数也是多项式范围的。

5.2.5 复杂度约束

由引理 3 我们知道，对于一个 Well-Type 的程序的指令 C，它的只可能有三种，$(0,0),(1,0),(1,1)$。下面将依次证明属于这三个 Type 的指令均能在多项式时间内计算。

引理 10 *对任何一个 Well-Type 的程序的指令 C，如果 $C:(0,0)$，那么 C 的计算时间不会超过多项式复杂度。*

证明 10 *如果 $C:(0,0)$，那么 C 中不会出现 while 语句，计算时间是常数。*

引理 11 *对任何一个 Well-Type 的程序的指令 C，如果 C 能够终止且 $C:(1,0)$，那么 C 的计算时间不会超过多项式复杂度。*

证明 11 *由引理 7，任何一个能够终止的指令在推导过程中不可能出现两个相同的 1 型状态，所以循环次数小于可能出现的 1 型状态数；又由引理 9，可能出现的 1 行状态数十多项式级别的，所以整个计算时间也是多项式级别的。*

引理 12 *对任何一个 Well-Type 的程序的指令 C，如果 C 能够终止且 $C:(1,1)$，那么 C 的计算时间不会超过多项式复杂度。*

证明 12 *考虑 C 中所有对 1 型变量的赋值语句 $X := E$，假设 C' 是 C 的最外层循环，由 Typing Rules 可知，$C':(1.0)$，所以如果 $E:(1,1)$，那么 E 一定不会出现在 C' 内部，进而 $X := E$ 的执行次数是常数次。有半安全运算的定义，存在一个常数 c 使得 $|[op](d_1,\cdots,d_n)| \leqslant max(d_i)+c$。那么在进入 C' 前，1 型变量的 size 至多增加常数次。而由上一条引理，在循环内的执行速度是最大 size 的多项式，进而整个执行的复杂度也是原来最大 size 的多项式。*

6 总结与拓展

综上，我们设计了一个针对能够约束命令式编程的复杂度的类型系统，在该系统下每一个 Well-Type 的程序都能够在多项式时间内得到计算，与此同时，每一个在图灵机上多项式时间内可解的程序也能够在我们的类型系统上用一个 Well-Type 的程序模拟。代码上，我们实现了整个系统的类型检查部分，没有实现求值部分。

我们认为这个类型系统具有一些可扩展点，有一些也想过但由于时间约束并没有实现，这些扩展点包括：

1. 与现有命令式编程语言相结合。将现有的 Type 与传统的 Type 相结合，使得复杂度的约束和类型检查可以同时进行。并进一步支持更多语法。
2. 能够自动化的推导出输入变量的类型，而不需要程序员指定。
3. 进一步的优化类型系统，使得 Well-Type 的程序一定能够终止。

7 人员分工

- 史杨勋惟：资料整理，系统设计与证明，报告撰写与展示。
- 蔡思培：系统设计，代码实现，样例测试。

8 参考资料

Marion, Jean-Yves, A Type System for Complexity Flow Analysis, LICS 2011