中国科学 E辑：信息科学 2008 年 第 38 卷 第 2 期：161 ~ 176
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE IN CHINA PRESS

一种新型形式化程序切片方法

张迎周[①*]，徐宝文[②]

① 南京邮电大学计算机学院，南京 210003;
② 东南大学计算机科学与工程学院，南京 210003
* E-mail: zhangyz@njupt.edu.cn

收稿日期: 2006-08-28; 接受日期: 2007-04-13
国家杰出青年基金(批准号: 60425206)、国家自然科学基金重点项目(批准号: 60633010)和国家自然科学基金青年基金(批准号: 60703086)资助项目

摘要 针对目前程序切片方法较单一，且其模块性和程序语言适应性较差，提出一种新型的形式化程序切片方法——基于模块单子语义的切片方法. 通过设计切片单子转换器，切片这一类计算被抽象成独立于具体语言的切片单子转换器，它可模块化地加载到实际程序中，得到相应的模块单子切片算法. 这种模块化的抽象机制使得单子切片算法具有较强的模块性和语言适应性. 还给出切片单子转换器的若干性质，并证明模块单子切片算法的正确性和终止性，同时研究单子切片算法与基于图可达性切片算法间的联系.

关键词
程序切片
单子
模块化
切片单子转换器
形式化语义

程序切片是由Weiser提出的一种重要的程序分析理解方法，用于从源程序中抽取对程序中兴趣点上的特定变量有影响的语句和谓词，组成新的程序(称作切片)，然后通过分析切片来分析源程序的行为. 程序切片分为两种: 动态切片和静态切片. 它广泛应用于程序分析、理解、调试、测试、软件维护、度量、逆向工程、再工程等领域[1~6].

目前人们已提出了多种程序切片方法，常用的主要有: 基于数据流方程的算法[7]和基于程序依赖图的图可达性算法[8]. 此外，还有一些针对不同应用的扩充算法. 但是，目前程序切片的算法还比较单一，基本上还是采用基于依赖图的图可达性算法[4]. 现有的切片方法是顺序增量式的，非复合式的，从而模块性较差. 但现代程序语言多数都支持模块化设计，其程序由一些模块构成，因此其程序分析方法(包括程序切片)应能反映该模块化设计特性.

动态切片据特定的输入来分析程序执行中的依赖关系，因其结果小且精确，在程序调试、错误定位等方面有着较高的应用价值. 动态切片的难点在于运行时信息的获取，为此，多数动态切片方法借助于关系图(如控制流图、依赖图等)，并追踪程序的执行过程，故不可避免地需要相当大内存空间来记录执行历史. 于是，动态切片技术研究的重点是如何减少追踪的信息

量.

Hwang等人曾指出程序切片受该程序语言的语义特性影响, 并证明了程序切片算法是语言相关的 [9], 所以切片算法应具有较强的灵活性来适应所给定语言的语义. 目前基于程序语义的切片方法主要是基于传统指称语义的程序切片, 即指称切片(denotational slicing)[10~12]. 但由于传统指称语义缺乏模块性和重用性 [13~15], 所以指称切片方法很难扩展应用到包含面向对象、并发等特性的实际语言中.

为此, 我们曾初步提出了一种新的语义切片方法: 模块单子切片 [16, 17], 它基于程序的模块单子语义(modular monadic semantics)[18]. 通过将程序切片这类计算抽象成独立于具体语言的切片单子转换器, 并将之模块化地加载到实际程序语义描述中, 形成相应的单子切片算法. 这样, 可直接在抽象语法项上计算切片, 无需在切片器中构造诸如控制流图或依赖图的中间结构; 在计算动态切片时也不必记录程序执行历史. 另外, 这种模块化抽象机制将使得相应的单子切片算法具有较强的重用性和程序语言适应性.

在文献 [16, 17]基础上, 本文将重新设计切片单子转换器, 并给出一些相关性质和证明. 第 1 节介绍模块单子语义的基本知识; 第 2 节给出新的切片单子转换器定义及其性质; 第 3 节介绍模块单子切片算法和一个实例; 第 4 节研究单子切片算法与基于依赖图切片算法间的关系, 指出单子切片算法所得结果与依赖图切片定义吻合, 并说明单子动态切片算法是精确的; 第 5 节给出单子切片算法正确性和终止性的证明; 第 6 节介绍算法实现情况, 并分析算法复杂度; 第 7 节进行相关实验结果分析; 第 8 节给出与相关工作的比较; 第 9 节总结全文并指出将来工作方向.

1 预备知识

单子(monad)概念来源于哲学, 是构成物质世界存在的最基本单位. 1950 年代单子已作为范畴论里一种函子, 但直到 1989 年才由Moggi将之引入语义描述框架中. 随后, Wadler将Moggi的单子方法广泛推广到函数式程序设计(尤其是Haskell语言)中 [19~21]. 采用单子这种更加规则的表示法替代 λ 表示法, 可避免定义语义结构时对无关语义成分的引用, 从而加强了语义描述的模块性和扩展性.

定义 1 一个**单子** M 是一个三元组$(m, return_m, bind_m)$, 包括类型构子 m 和两个多态函数:

$$return_m :: a \to m\ a,$$

$$bind_m :: m\ a \to (a \to m\ b) \to m\ b,$$

且它们满足如下规律 [19, 20]:

左幺元: $(return_m\ a)\ `bind_m`\ k = k\ a$,

右幺元: $e\ `bind_m`\ return_m = e$,

结合律: $e\ `bind_m`\ (\lambda a.(k\ a)\ `bind_m`\ (\lambda b.h\ b)) = (e\ `bind_m`\ (\lambda a.k\ a))\ `bind_m`\ (\lambda b.h\ b)$.

直观上, 对于类型 a, $m\ a$ 则表示所有能得到 a 类型结果的计算, 这些计算在范畴理论的同态概念下是一致的. 单子所要满足的 3 个规律说明: 函数 $return_m$ 有左右单位元, 其相应的空计算可跳过特定的上下文; 函数的组合是满足结合律的, 它可捕获顺序程序执行中最基本的特性. 为方便起见, 后面采用如下的简写形式:

$$\{e\} \equiv e;\ \{m;\ e\} \equiv m\ `bind_m'\ \lambda\ \_.\ \{e\};\ \{x \leftarrow m;\ e\} \equiv m\ `bind_m'\ \lambda\ x.\ \{e\}.$$

定义 2 单子转换器由类型构造器 t 及其提升函数 $lift_t$ 构成, 其中 t 将给定的单子$(m,\ return_m,\ bind_m)$映射到新单子$(t\ m,\ return_{t\ m},\ bind_{t\ m})$, 提升函数 $lift_t$:

$$lift_t:\quad m\ a \to t\ m\ a,$$

还要满足

$$lift_t\ return_m = return_{t\ m},$$
$$lift_t\ (e\ `bind_m'\ k) = (lift_t\ e)\ `bind_m'\ (lift_t\ k).$$

单子转换器不仅提供了一种抽象化表示程序特性的能力, 而且还允许人们访问低层语义的细节部分. 概念"提升"使得我们能考虑不同程序特性间的交互. 单子转换器的两个规律保证了提升的基本性质: 对任何一个程序, 如果不使用由单子转换器所引入的新特性, 则在单子转换器作用前后, 该程序的表现行为应该一致.

因单子转换器完全独立于具体语言, 仅表示某类计算, 故单子转换器的设计可认为是一劳永逸的. 目前人们已设计了不少单子转换器 [18,19,22,23]. 我们在文献 [16]中设计了切片单子转换器, 以此来统一描述程序切片这类计算. 本文将重新设计该切片单子转换器, 见第 2 节.

模块单子语义通过将语法项映射到计算(单子)来形式化描述程序语义. 模块单子语义的关键特征是: 多个单子转换器可结合到一个单子中, 从而使得最终单子包含所有要描述的程序概念.

为本文讨论方便, 考虑一个简单命令式语言 W, 其抽象语法和模块单子语义见图 1. 我们假设图 1 中的表达式没有诸如赋值等的副作用, 并赋予每个表达式一个唯一标号. 图 1 中, 符

论域:

ide: Ide (标识符); l: Label (标号); e: Exp (表达式); *loc*: Loc (存储); *v*: Value (值域)

抽象语法:

S :: = ide := l.e | S_1; S_2 | skip | **if** l.e **then** S_1 **else** S_2 **endif** | **while** l.e **do** S **endwhile**

语义方程:

$$\boldsymbol{S}:: \text{Stmt} \to \text{ComptM}\ ()$$
$$\boldsymbol{E}:: \text{Exp} \to \text{ComptM Value}$$

$$\boldsymbol{S}[\![\text{ide} := \text{l.e}]\!] = \{v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];\ loc \leftarrow lkpEnv(\text{ide}, rdEnv);\ updSto(loc, return\ v)\}$$
$$\boldsymbol{S}[\![S_1;\ S_2]\!] = \{\boldsymbol{S}[\![S_1]\!];\ \boldsymbol{S}[\![S_2]\!]\}$$
$$\boldsymbol{S}[\![\text{skip}]\!] = return\ ()$$
$$\boldsymbol{S}[\![\textbf{if}\ \text{l.e}\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}]\!] = \{v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];\ \textbf{case}\ v\ \textbf{of}\ \ \textbf{tt} \to \boldsymbol{S}[\![S_1]\!];\ \textbf{ff} \to \boldsymbol{S}[\![S_2]\!]\ \ \}$$
$$\boldsymbol{S}[\![\textbf{while}\ \text{l.e}\ \textbf{do}\ S\ \textbf{endwhile}]\!] = \boldsymbol{Fix}\ (\lambda f.\{v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];\ \textbf{case}\ v\ \textbf{of}\ \ \textbf{tt} \to f \cdot \boldsymbol{S}[\![S]\!];\ \textbf{ff} \to return\ ()\ \ \}\ )$$

图 1 实例语言 W 的模块单子语义

号***Fix***表示不动点算子; *lkpEnv*是环境Env的查找操作(函数); *updSto*是存储Loc的更新操作; *getValue*和*putValue*是I/O单子的基本操作[18,23]. 最终单子ComptM可由基本单子(如输入/输出单子IO或恒等单子Id)和作用其上的单子转换器构成, 如

$$\text{ComptM} \equiv (\text{StateT} \cdot \text{EnvT})\ \text{IO}.$$

为后面讨论切片的需要, 我们给出 W 语言的 *Syn*(s, *L*)定义(见图 2), 其中 s 表示被分析的 W 源程序; *Refs*(l.e)表示所有出现在表达式 l.e 中变量的集合; ε 表示空操作. *Syn*(s, *L*)说明了如何根据所求得的切片表中标号集合 *L*, 从源程序 s 中构建一个符合文法的 W 子程序, 即相应的程序切片. *Syn*(s, *L*)允许我们只需关注所分析程序中的加标表达式, 因为程序切片的主体部分依赖于语句中的表达式, 其他部分可由 *Syn*(s, *L*)捕获.

Syn(s, *L*) =
  case s **of**
    "ide := l.e " : **if** l $\in$ *L* **then** "ide := l.e" **else** ε
    "S_1; S_2 " : *Syn*(S_1, *L*); *Syn*(S_2, *L*)
    "skip " : ε
    "**if** l.e **then** S_1 **else** S_2 **endif** " : **if** (*Syn*(S_1, *L*) = *Syn*(S_2, *L*) = ε)$\wedge$(l $\notin$ *L*) **then** ε
      else "**if** l.e **then** *Syn*(S_1, *L*) **else** *Syn*(S_2, *L*) **endif** "
    "**while** l.e **do** *S* **endwhile** " : **if** (*Syn*(*S*, *L*) = ε)$\wedge$(l $\notin$ *L*) **then** ε
      else "**while** l.e **do** *Syn*(*S*, *L*) **endwhile** "

图 2 *Syn*(s, *L*)的定义

2 切片单子转换器

本节将重新设计文献 [16, 17]中的切片单子转换器SliceT, 见图 3 所示. 图 3 中*为单元类型Unit中唯一的元素; *L*表示进行当前语句计算所需表达式的标号集合; *s*为切片表Slices, 其数据结构如下:

type Var = String
type Labels = [Int]
type Slices = [(Var, Labels)]
lkpSli :: Var $\rightarrow$ Slices $\rightarrow$ ComptM Labels
updSli :: (Var, ComptM Labels) $\rightarrow$ Slices $\rightarrow$ ComptM ()
mrgSli :: Slices $\rightarrow$ Slices $\rightarrow$ ComptM Slices

每个变量所对应的切片是一个标号集合, 所有变量及其切片构成了一张表(Hash 表), 记为切片类型 Slices. 它包括 3 个基本的操作函数: *lkpSli*, *updSli* 和 *mrgSli*, 分别用来查找 Slices 中某变量对应的切片(标号集合)、更新 Slices 中数据和合并两个 Slices.

rdLabels, *inLabels*和*getSli*, *setSli*分别是SliceT中参数*L*和*s*的读取和更新操作. 在文献 [16, 17]中, SliceT只显示地接收参数*L*, 故需额外增加Slices的相应操作来反映切片表变化. 图 3 的SliceT中, 我们将*L*和*s*均作为其参数, 从而SliceT可直接显示地反映切片表的变化.

至于图3中单子转换器SliceT *L s m*定义的正确性, 可由下面两个定理来保证, 其证明类似文献 [17]中定理证明, 故在此省略.

$$
\begin{aligned}
&\textbf{type}\ \text{SliceT}\ L\ s\ m\ a = (L, s) \to m\ (a, s)\\
&return_{\text{SliceT}\ L\ s\ m}\ \ x \ \ = \lambda(L, s).\ return_m\ (x, s)\\
&e\ `bind_{\text{SliceT}\ L\ s\ m}`\ f \ \ = \lambda(L, s).\ \{(a, s') \leftarrow e\ (L, s)\ ;\ f\ a\ (L, s')\}_m\\
&lift_{\text{SliceT}\ L\ s}\ \ e \qquad = \lambda(L, s).\ \{a \leftarrow e\ ;\ return_m\ (a, s)\}_m\\
&\qquad rdLabels \qquad = \lambda(L, \_).\ return_m\ (L, *)\\
&\qquad inLabels\ L\ c \ \ = \lambda(L', s).\ c\ (L, s)\\
&\qquad getSli \qquad\ \ = \lambda(\_, s).\ return_m\ (s, s)\\
&\qquad setSli\ \ s \qquad = \lambda(\_, s').\ return_m\ ((), s)
\end{aligned}
$$

图 3　切片单子转换器 SliceT

定理 1　图 3 中所定义的 SliceT *L s m* 是一个单子.

定理 2　图 3 中所定义的 SliceT *L s* 是一个单子转换器.

公理　关于切片转换器 SliceT 中的操作 *getSli*, *rdLabels* 和 *inLabels*, 以及切片表 Slices 中的操作 *updSli* 和 *lkpSli*, 有如下的一些等式关系, 其中 *x* :: Var, *z* :: *a*, *e* :: *M a*(*a* 为任一类型), *L*, *ls* :: Labels, *s* :: Slices, *f* 为函数:

(1) $updSli(x, L, getSli) = \{s \leftarrow getSli;\ updSli(x, L, s)\}$,
$lkpSli\ (x, getSli) = \{s \leftarrow getSli;\ lkpSli\ (x, s)\}$.

(2) $\{updSli(x, lkpSli(x, getSli), getSli);\ getSli\} = getSli$.

(3) $\{updSli(x, L, getSli);\ lkpSli(x, getSli)\} = \{updSli(x, L, getSli);\ return\ L\}$.

(4) $\{s \leftarrow getSli;\ s' \leftarrow getSli;\ f\ (s, s')\} = \{s \leftarrow getSli;\ s' \leftarrow getSli;\ f\ (s, s)\}$.

(5) $\{s \leftarrow getSli;\ updSli(x, L, s');\ f\ s\} = \{updSli(x, L, s');\ f\ (getSli)\}$.

(6) $\{updSli(x, L, s);\ updSli(x', L', s')\} = \{updSli(x', L', s');\ updSli(x, L, s)\}$.

(7) $\{v \leftarrow return\ z;\ updSli(x, L, getSli);\ f\ v\} = \{updSli(x, L, getSli);\ f\ (return\ z)\}$.

(8) $\{ls \leftarrow rdLabels;\ updSli(x, L, getSli);\ f\ ls\} = \{updSli(x, L, getSli);\ f\ (rdLabels)\}$.

(9) 如果 $\{v \leftarrow e;\ updSli(x, L, getSli);\ f\ v\} = \{updSli(x, L, getSli);\ v \leftarrow e;\ f\ v\}$, 则有
$\{v \leftarrow inLabels\ ls\ e;\ updSli(x, L, getSli);\ f\ v\} = \{updSli(x, L, getSli);\ v \leftarrow inLabels\ ls\ e;\ f\ v\}$.

(10) $\{v \leftarrow return\ z;\ s \leftarrow getSli;\ f\ (v, s)\} = \{s \leftarrow getSli;\ v \leftarrow return\ z;\ f\ (v, s)\}$.

(11) $\{ls \leftarrow rdLabels;\ s \leftarrow getSli;\ f\ (ls, s)\} = \{s \leftarrow getSli;\ ls \leftarrow rdLabels;\ f\ (ls, s)\}$.

(12) 如果 $\{v \leftarrow e;\ s \leftarrow getSli;\ f\ (v, s)\} = \{s \leftarrow getSli;\ v \leftarrow e;\ f\ (v, s)\}$, 则有
$\{v \leftarrow inLabels\ ls\ e;\ s \leftarrow getSli;\ f\ (v, s)\} = \{s \leftarrow getSli;\ v \leftarrow inLabels\ ls\ e;\ f\ (v, s)\}$.

公理(3)说的是, 以标号集合 *L* 更新当前切片表中变量 *x* 的切片, 再查找 *x* 的切片, 这等价于以 *L* 更新当前切片表中 *x* 的切片, 并直接返回 *L*; 公理(4)表示连续两次的 *getSli* 操作返回结果是一样的; 公理(5)~(9)指出, 在某些情形下, 更新操作 *updSli* 可与其他操作更换执行次序; 公理(10)~(12)说明操作 *getSli* 有时候与执行次序无关.

3　模块单子切片算法

在文献 [16, 17]中, 我们给出了程序切片的模块单子算法, 包括单子静态切片算法和动态

切片算法, 见算法 1 和 2. 算法中仅考虑程序最终点的单变量程序切片, 对应静态和动态切片标准分别为: ⟨p, v⟩和⟨INPUT, p, v⟩, 其中INPUT为用户所关心的某个输入集; p为程序最后语句点; v为程序中某个变量. 算法 2 中, 因最终将得到程序中所有单个变量的单子切片, 所以动态切片标准可写成⟨INPUT, p⟩. 通过合并相应单个变量的程序切片, 可得到多个变量的程序切片 [2,24].

输入: 静态切片标准⟨p, v⟩
输出: 静态切片
5. 初始化标号集合 L 和 Slices 中标号集合, 一般为空集.
6. 将切片单子转换器组合到程序语义模块中.
7. 按模块单子方法逐条语句计算相应变量切片, 得到包含所有变量切片的 Slices.
8. 根据 Slices 和 Syn(s, *L*)的定义返回最终静态切片.

算法 1　模块单子静态切片算法

输入: 动态切片标准⟨INPUT, p⟩
输出: 所有单个变量的动态切片
1. 初始化标号集合 *L* 和 Slices 中标号集合.
2. 将切片单子转换器组合到程序语义模块中.
3. 以 INPUT 作为输入, 按模块单子方法执行被分析程序, 同时得到了包含所有变量切片的 Slices.
4. 根据 Slices 和 Syn(s, *L*) 的定义返回相应动态切片.

算法 2　模块单子动态切片算法

模块单子切片算法的基本思想为: 先将切片单子转换器组合到语义模块描述中, 使其包含程序切片计算, 然后按此语义描述逐句分析源代码, 或根据特定的输入执行源程序, 最后得到所要求的单子切片.

为了将切片计算模块化地加到程序分析中, 可通过如下方式将 SliceT 组合到图 1 里的最终单子 ComptM 中, 其组合的次序可以任意:

$$\text{ComptM} \equiv (\text{SliceT} \cdot \text{StateT} \cdot \text{EnvT})\ \text{IO}.$$

在语义描述中, 若其中含有某个加标表达式 l.e 的计算, 则将原来的标号集合 L 转变成如下的中间集合 L', 以此记录下相应的依赖关系:

$$L' = \{\text{l}\} \cup L \cup \bigcup_{r \in \text{Re}\, fs(\text{l.e})} lkpSli\ (r, getSli)\,. \tag{1}$$

(1)式反映了影响当前计算的表达式标号集合的变化情况, 即如果当前计算中包含表达式 l.e 的计算, 则要将该表达式标号以及影响其中变量的所有标号集合合并到计算前的标号集合中, 从而形成新的标号集合, 并将之传递下去.

例如本文 W 语言的赋值语句、条件语句和循环语句的静态切片语义描述分别为

$$\boldsymbol{S}[\![\text{ide} := \text{l.e}]\!] = \{L \leftarrow rdLabels;\ v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];\ L' \leftarrow \{\text{l}\} \cup L \cup \bigcup_{r \in \text{Re}\, fs(\text{l.e})} lkpSli\ (r, getSli)\ ;$$
$$loc \leftarrow lkpEnv(\text{ide}, rdEnv);\ updSto(loc, return\ v);\ updSli(\text{ide}, L', getSli)\}$$

$$\boldsymbol{S}[\![\textbf{if}\ \text{l.e}\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}]\!] = \{\ L \leftarrow rdLabels;\ L' \leftarrow \{\text{l}\} \cup L \cup \bigcup_{r \in \text{Re}\, fs(\text{l.e})} lkpSli\ (r, getSli)\ ;$$
$$\text{T} \leftarrow getSli;\ inLabels\ L'\ \boldsymbol{S}[\![S_1]\!];\ \text{T1} \leftarrow getSli;\ setSli(\text{T});$$
$$inLabels\ L'\ \boldsymbol{S}[\![S_2]\!];\ \text{T2} \leftarrow getSli;\ mrgSli(\text{T1}, \text{T2})\}$$

$$S[\![\textbf{while}\ \text{l.e}\ \textbf{do}\ S\ \textbf{endwhile}]\!] = \textit{Fix}\ (\lambda f.\ \{\ L \leftarrow rdLabels;\ L' \leftarrow \{\text{l}\} \cup L \cup \bigcup_{r \in \text{Re}\,fs(\text{l.e})} lkpSli\ (r, getSli)\ ;\ \text{T} \leftarrow getSli;\ f \cdot \{inLabels\ L'\ S[\![S]\!];\ \text{T}' \leftarrow getSli;\ mrgSli(\text{T}, \text{T}')\ \}\})$$

例如, 对图 4 所示的源程序, 其表达式的标号为该表达式所在语句片断的标号(如可取相应的行号), 于是第 4 号表达式为 i < 10. 经过第 3 号表达式后, 该表达式的初始标号集合 L 转变为

$$L' = \{3\} \cup L \cup lkpSli\ (\text{i}, getSli) = \{3\} \cup \phi \cup \phi = \{3\}$$

又因 3 号表达式是赋值语句, 故变量 i 的切片(简记为 L(i))被替换为 L', 即{3}. 类似地, 对于第 4 号表达式有

```
3      i := 0 ;
4      while i < 10 do
5        if i == 0 then
6          s:=10
         endif ;
8        i := i + 1;
       endwhile
```

图 4　一个实例程序

$L' = \{4\} \cup L \cup lkpSli\ (\text{i}, getSli) = \{4\} \cup \phi \cup \text{L(i)} = \{3, 4\}$

5 号: $L'' = \{5\} \cup L' \cup \text{L(i)} = \{3, 4, 5\}$

6 号: $L''' = \{6\} \cup L'' \cup \text{L(s)} = \{3, 4, 5, 6\}$; L(s)={3, 4, 5, 6}

8 号: $L'' = \{8\} \cup L' \cup \text{L(i)} = \{3, 4, 8\}$; L(i)={3, 4, 8}

因循环, 再次分析 4 号表达式后,

$L'' = \{4\} \cup L' \cup \text{L(i)} = \{3, 4, 8\}$

5 号: $L''' = \{5\} \cup L'' \cup \text{L(i)} = \{3, 4, 5, 8\}$

6 号: $L'''' = \{6\} \cup L''' \cup \text{L(s)} = \{3, 4, 5, 6, 8\}$; L(s)={3, 4, 5, 6, 8}

8 号: $L''' = \{8\} \cup L'' \cup \text{L(i)} = \{3, 4, 8\}$; L(i)={3, 4, 8}

合并两次计算的结果

$$\text{L(s)} = \{3, 4, 5, 6\} \cup \{3, 4, 5, 6, 8\} = \{3, 4, 5, 6, 8\};\quad \text{L(i)} = \{3, 4, 8\}$$

此时若再循环分析下去, i 和 s 的切片不再变化, 即达到了不动点, 可得最终点处变量 s 和 i 的静态切片(主体部分)为

$$\text{L(s)} = \{3, 4, 5, 6, 8\};\quad \text{L(i)} = \{3, 4, 8\}.$$

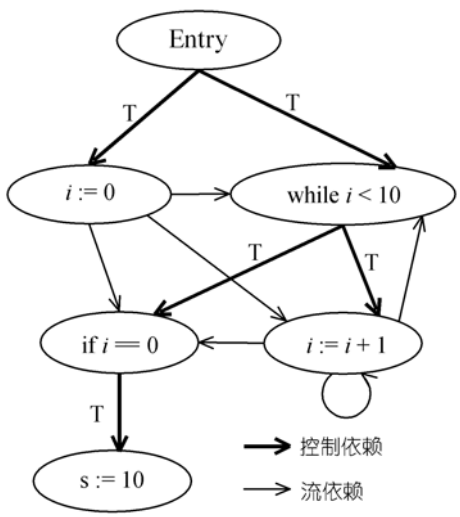

图 5　图 4 中程序的依赖图

关于上述算法 1 和 2 正确性的证明见本文第 5 节. 证明之前先说明单子切片算法所得到的结果与依赖图的切片定义吻合.

4　算法结果与依赖图切片定义吻合

程序P的程序依赖图是一个有向图$G\ (V, E)$, V为顶点集, 表示程序组成成分, 一般为程序中出现的赋值语句和控制断言; E为边集, 表示程序成分间的依赖关系, 从顶点v_1 到顶点v_2 的依赖关系包括控制依赖, 记为$v_1 \rightarrow_c v_2$, 和数据依赖(或流依赖), 记为$v_1 \rightarrow_f v_2$, 并称v_1 为该边的源点, v_2 为目标点 [24~26]. 控制依赖边上常标有T或F(表示true或false), 其源顶点一般是Entry顶点和断言顶点. 如图 4 中实例程序的依赖图见图 5 所示.

一个程序依赖图 G 关于顶点 s 的切片, 记为 G/s, 是包含能通过流依赖边和控制依赖边到达 s 的所有顶点的图. 顶点集 $V(G/s) = \{w \mid w \rightarrow^*_{c,f} s, w \in V(G)\}$. 这可扩展到集合 $S = \bigcup_i s_i$ 的定义:

$$V(G/S) = V(G/(\bigcup_i s_i)) = \bigcup_i V(G/s_i).$$

对某个顶点 v, 如果 $v \notin G$, 则定义 $V(G/v) = \phi$. 图 G/S 的边主要是由顶点集 $V(G/S)$所产生的 G 子图, 即有

$$E(G/S) = \{(v \rightarrow_f w) \mid (v \rightarrow_f w) \in E(G) \wedge v, w \in V(G/S)\} \cup \{(v \rightarrow_c w) \mid (v \rightarrow_c w) \in E(G) \wedge v, w \in V(G/S)\}.$$

就本文所考虑的简单语言 W 而言, 其程序依赖图中的顶点, 除了入口顶点 Entry 外, 要么是判定谓词, 要么就是赋值语句, 都有相应的加标表达式与之对应. 于是, 每个顶点 w 有唯一的加标表达式 l_w.e, 对应唯一的标号 l_w.

在单子切片算法中, 每执行/分析一次加标表达式时, 就按(1)式计算一次相应的中间标号集合 L', 如

$$L_w' = \{\mathrm{l}_w\} \cup L_w \cup \bigcup_{r \in \mathrm{Re}\, fs(\mathrm{l}_w.\mathrm{e})} lkpSli\ (r, getSli),$$

且对于同一顶点来说, 其中间标号集合会依其计算次数递增, 如 $L_w' \subseteq L_w'' \subseteq L_w''' \subseteq L_w'''' \subseteq \cdots \subseteq L_w$, 其中 L_w 为顶点 w 最后一次伴随 l_w.e 计算所产生的中间标号集合.

对于可达的两个顶点 w 和 $v(w \rightarrow^*_{c,f} v)$, 存在从 w 到 v 的可执行路径, 使得在该路径上只要 w 被执行/分析, v 也一定会被执行/分析. 为简单起见, 在下面叙述中, 如不加说明, 可达顶点(如 w 和 v)及其可执行路径上中间节点(如 u)的中间标号集合, 都是在同一次执行/分析该可执行路径时所产生的, 统记为 L_w', L_v'和 L_u'.

引理 1 如果依赖图中顶点 w 与 v 直接可达, 即 $w \rightarrow_{c,f} v$, 则 $L_w' \subseteq L_v'$.

证明 分两种情形证明.

1) w 为一个谓词, 即 $w \rightarrow_c v$, 则 L_w'将作为 L_v'中的 L_v 传递到 v, 即 $L_v = L_w'$, 于是 $L_w' \subseteq L_v'$.

2) w 是一个(对某变量 x)赋值语句, 即 $w \rightarrow_f v$, 此时变量 x 的切片 $\mathrm{L}(x) = L_w'$且 $x \in \mathrm{Re}fs(\mathrm{l}_w.\mathrm{e})$, 则 L_v'中的$\bigcup_{r \in \mathrm{Re}\, fs(\mathrm{l}_v.\mathrm{e})} lkpSli\ (r, getSli)$ 必然会包含 L_w', 于是 $L_w' \subseteq L_v'$.

引理 2 如果依赖图中顶点 w 与 v 可达, 则顶点 w 所对应的标号 l_w 一定包含于 L_v'. 即若 $w \rightarrow^*_{c,f} v$, 则 $\mathrm{l}_w \in L_v'$.

证明 分两种情形证明.

1) 顶点 w 与 v 直接可达: $w \rightarrow_{c,f} v$. 于是 w 可通过流依赖边或控制依赖边直达 v, 如果 $w \rightarrow_f v$, 这说明 w 是一赋值语句, 且 l_v.e 引用了 w 中定义的某个变量(如 y), 即: $\mathrm{l}_w \in \mathrm{L}(y)$且 $y \in \mathrm{Re}fs(\mathrm{l}_v.\mathrm{e})$, 则 L_v'中的$\bigcup_{r \in \mathrm{Re}\, fs(\mathrm{l}_v.\mathrm{e})} lkpSli\ (r, getSli)$ 必然会包含 l_w; 如果 $w \rightarrow_c v$, 即 w 为一个谓词, 则 l_w 必然会包含到 L_v'中的 L_v 里. 故该情形下, 有 $\mathrm{l}_w \in L_v'$.

2) w 与 v 间接可达: $w \rightarrow^+_{c,f} v$. 如果 w 先直达某个顶点 u, u 再直达顶点 v, 即 $w \rightarrow_{c,f} u \rightarrow_{c,f} v$, 则由第 1)种情形知, l_w必然会进入 u 对应的 L'(记为 L_u')里, 即 $\mathrm{l}_w \in L_u'$. 而又由引理 1 知, $L_u' \subseteq L_v'$,

从而当 $w \to_{c,f} u \to_{c,f} v$ 时, $l_w \in L_v'$. 类似地分析下去有: 如果 $w \to^{+}_{c,f} v$, 则 $l_w \in L_v'$.

定理 3 单子切片算法可捕获相应程序依赖图中的依赖关系.

证明 对于依赖图中的流依赖关系 $w \to_f v$, 单子切片算法通过 v 的 L_v'中

$$\bigcup_{r \in \mathrm{Re}fs(l_v.e)} lkpSli\ (r, getSli)$$

捕获. 另外, L_v'中标号集合 L_v 可包含相应的控制依赖关系 $w \to_c v$.

具体地, 由引理 2 知, 对于两个可达的顶点 w 与 v(即 $w \to^{*}_{c,f} v$), 有 $l_w \in L_v'$, 又 $L_v' \subseteq L_v$, 从而 l_w 必然会包含入 v 的最终中间标号集合里. 所以, 如果 w 由依赖图的可达性进入关于 v 中某个变量的切片, 则单子切片算法也能捕获到该顶点(标号).

定理 4 按依赖图的切片定义, 动态单子切片算法(算法 2)在精度上是精确的.

证明 由定理 3 知, 依赖图中相应可达信息被单子切片算法(包括算法 2)捕获了. 现只要证明算法 2 中的结果不含多余的依赖信息, 即要证: 如果顶点 v 处的 L_v'包含了某个标号 l_w($v \neq w$), 则相应依赖图中 w 与 v 可达. 因为 $l_w \in L_v'$, 则 $l_w \in L_v$ 或者

$$l_w \in \bigcup_{r \in \mathrm{Re}fs(l_v.e)} lkpSli\ (r, getSli)\ .$$

若 $l_w \in L_v$, 则存在顶点 v_1, 使得 $v_1 \to_c v$, 且 $l_w \in L_{v_1}'$; 若 $l_w \in \bigcup_{r \in \mathrm{Re}fs(l_v.e)} lkpSli\ (r, getSli)$ 中, 则存在对某变量 x_1 赋值的顶点 w_1, 使得 $w_1 \to_f v$, $x_1 \in \mathrm{Re}fs(l_v.e)$, $L(x_1) = L_{w_1}'$, 且 $l_w \in L_{w_1}'$. 对顶点 v_1(或 w_1)类似地分析, 存在顶点 v_2, 使得 $v_2 \to_{c,f} v_1$, 且 $l_w \in L_{v_2}'$ (或存在顶点 w_2, 使得 $w_2 \to_{c,f} w_1$, 且 $l_w \in L_{w_2}'$). 依次下去, 直到顶点 $v_n = w$(即 $l_{v_n} = l_w$), 且 $v_n \to_{c,f} v_{n-1} \to_{c,f} v_{n-2} \to_{c,f} \dots \to_{c,f} v_1 \to_c v$ (或 $w_n = w$, 且 $w_n \to_{c,f} w_{n-1} \to_{c,f} w_{n-2} \to_{c,f} \dots \to_{c,f} w_1 \to_f v$), 于是 $w \to^{*}_{c,f} v$, 即 w 与 v 可达.

5 单子切片算法正确性的证明

模块单子切片算法的本质为: 随着程序的执行或分析, 根据语句的切片语义描述规则, 切片表 Slices(初始为空)被不断地更新. 由定理 3 知, 单子切片算法可捕获相应的依赖关系, 其所得结果与基于依赖图的切片定义相吻合. 下面就 W 语言的语句进行结构归纳, 证明算法 1 和 2 的正确性.

定理 5 就 W 语言而言, 单子切片算法 1 和 2 是正确的.

证明 对 W 语言的语句 S 进行结构归纳证明.

(1) 赋值语句 x := l.e .

在程序依赖图中, 赋值语句所在的顶点是流依赖边的源点. 相应地, 在单子语义切片中, 在对变量 x 赋值定义的同时, 通过操作 *updSli*(x, L', *getSli*)将伴随 l.e 计算所产生的中间标号 L' 保存为此时的 x 单子切片, 以便将之按流依赖边传递下去.

(2) 循环语句 **while** l.e **do** S **endwhile** .

While 循环语句中的谓词部分(**while** l.e)控制依赖语句 S, 且依赖边上标记为 true, 于是那些影响 l.e 计算的所有语句(即 L')均可通过该控制依赖边到达 S, 所以单子切片中语句 S 所对应的 L 被更新为 L'.

在动态单子切片(算法 2)中, While 循环的执行终止条件取决于谓词 **while** l.e, 如果循环不终止, 即程序本身不能正常终止, 则单子切片算法也不停机. 故可假设循环语句在第 j 次执行后终止. 现只需证明循环体 S 在第 j 次执行后, 其切片表 Slices 能正确捕获经 j 次循环后所有关于 S 中变量的可达信息(切片). 对执行次数进行归纳证明:

归纳基: 由先前的结构归纳假设知道, 循环体 S 在第 0 次执行时已正确捕获了所有 S 中变量的可达信息.

归纳步: 假设循环体 S 在第 i 次(i < j)执行后能正确捕获所有 S 中变量的可达信息, 现需证明 S 在第 i+1 次执行后结论成立. 用反证法证明:

如果 S 在第 i+1 次执行后未能正确捕获所有 S 中变量的可达信息, 则设 S 中变量 x 的可达信息未能正确捕获, 即在循环第 i+1 次执行后, 切片表 Slices 中 x 的切片(记为 $L_{i+1}(x)$)不正确. 也就是说, $L_{i+1}(x)$中至少未包含一个关于 x 可达的顶点 w, 即 $l_w \notin L_{i+1}(x)$. 根据假设, 切片 $L_i(x)$ 是正确的, 若 $L_{i+1}(x)$不正确, 则 S 的执行过程中必须至少执行了 1 次对 x 的赋值语句 v(如 $x := l_v.e$), 并到达 S 语句的末尾, 即相应单子切片中执行了对 x 切片的更新操作 $updSli(x, L_v', getSli)$, 且 $L_{i+1}(x) = L_v'$.

若 $L_{i+1}(x)$的不正确是因没有包含相应依赖图上关于 x 可达的顶点 w, 即 $l_w \notin L_{i+1}(x)$. 但由引理 2 知道, 如果 w 与 v 关于 x 可达, 则一定有 $l_w \in L_v' = L_{i+1}(x)$, 从而与假设矛盾.

从而由反证法知道, S 在第 i+1 次执行后也能正确捕获所有 S 中变量的可达信息. 所以根据结构归纳法, 循环体 S 在第 j 次执行后, 其切片表 Slices 能正确捕获经 j 次循环后所有关于 S 中变量的可达信息.

在静态单子切片(算法 1)中, While 循环的切片算法正确性证明与上述类似, 只是其分析终止条件为最终切片表的稳定. 由于每次循环后, 就合并循环前后的切片表, 即切片表数据随循环次数非减, 而循环中的语句是有限的, 这样最终切片表必然在有限次内(记为 j 次, 最多次数为该循环体所有加标表达式的数目)达到稳定.

(3) 条件语句 **if** l.e **then** S_1 **else** S_2 **endif** .

If 条件语句中的谓词部分(**if** l.e)控制依赖语句 S_1 和 S_2, 其依赖边分别标记为 true 和 false. 与循环语句类似, 伴随 l.e 的 L'被传递给 S_1 和 S_2, 即单子切片中语句 S_1 和 S_2 对应的 L 均被更新为 L'. 在动态单子切片中, 根据 l.e 的计算结果, 选择进入相应的分支, 即如果 l.e 的值为 true, 则 L'沿着标号为 true 的依赖边向下传递, 到达语句 S_1; 反之 L'沿标号为 false 的依赖边传递到 S_2. 在静态单子切片中, 因为要考虑所有可能的依赖, 故需将 L'同时传递给 S_1 和 S_2, 并将其切片表合并.

(4) 复合语句情形 $S_1; S_2$.

因为复合语句中 S_1 和 S_2 间不可能存在控制依赖, 即只可能是流依赖. 由结构归纳假设, S_1 和 S_2 都已经正确地捕获了相应的依赖边, 由引理 2 和定理 3 知, S_2 中一定成功捕获了 S_1 和 S_2 间可能存在的流依赖边.

(5) 空语句 skip: 结论显然成立.

定理 6(终止性定理) 对于一个合法的 W 程序 P, 如果 INPUT 为使得 P 正常终止的一次

输入, 则有: (1) 算法 1 总是收敛; (2) 算法 2 关于 INPUT 收敛.

证明 对于 W 语言, 只有 While 循环语句才可能导致单子切片算法不终止(发散).

(1) 在算法 1 中, While 语句分析的终止条件为最终切片表的稳定. 由定理 5 中(情形 2)分析知, 最终切片表可经有限次循环达到稳定.

(2) 在算法 2 中, While 语句的可终止性与该语句实际执行的可终止情形一致. 因程序 P 按 INPUT 执行终止, 所以其切片算法 2 也可终止(收敛).

6 算法实现及复杂度分析

6.1 算法实现

通过使用单子转换器,模块单子切片算法将具有较高的模块性和语言适应性, 它允许我们单独地处理每类计算, 然后靠模块单子系统简洁且安全地组合起来. 此外, 模块单子系统还是可执行的, 因为其语义描述具有清晰的操作解释. 目前已经存在了一些基于单子转换器的模块化编译器或解释器, 如文献 [23, 27]. 目前我们正在开发的单子切片器原型系统MSPS以程序的模块单子语义为基础, 结合共代数(coalgebraic)和类属(generic)概念, 支持增量式开发, 其实现语言为Haskell.

由 MSPS 系统对图 4 程序切片分析结果见图 6, 其中 var 语句用来声明变量, write 语句用来输出变量值; 结果中包含了分析/执行路径、最终静态/动态切片表和切片结果、整个过程(包括记录路径、输出结果)所耗费 CPU 的时间(本文中实验数据的运行机器基本配置为: Intel Pentium IV 2.6G 处理器、512M 内存). 另外, 程序执行后所输出的结果显示在动态切片信息中.

6.2 算法复杂性分析

由于模块单子编译器/解释器可自动、安全地将切片单子转换器模块化加载到语义模块中, 又由于切片表Slices是Hash表(其基本操作复杂度是常数), 所以本文单子切片算法时间复杂度分析主要是针对具体程序语言的L'和$Syn(s, L)$. 本文所定义的L'的时间复杂度为$O(v)$, 其中v为程序中单变量数目; 图 2 中$Syn(s, L)$的时间复杂度为$O(m)$, 其中m为程序中加标表达式的数目. 从而对于相应的单子切片算法(算法 1/算法 2), 其切片主体部分(不可执行的程序切片) 时间复杂度为$O(v \times n)$, 其中n为程序中被分析/执行的所有加标表达式(含重复)数目. 一般地, 在静态分析情形下, n与m呈线性关系. 另外, 为得到所有单变量的可执行切片, 需额外的时间复杂度为$O(v \times m)$. 因此本文单子切片算法的总时间复杂度为$O(v \times n + v \times m)$, 又因单子切片算法最终得到了所有单变量的单子切片, 故平均每个变量单子切片的时间复杂度为$O(n + m)$, 是线性的. 因为静态切片算法 1 是逐条语句分析源程序的, 自然有$n \geq m$, 所以静态单子切片算法的平均每个变量单子切片的时间复杂度可写为$O(n)$. 实验表明(包括考虑了文献 [2]中的特殊循环, 见第 7 节), n与m一般呈线性关系, 最坏情形下$n = O(m^2)$.

至于单子切片算法的空间复杂度分析, 主要是针对 Re*f*s(l.e), Slices, L'和 L. 需要 $O(v \times v)$和 $O(v \times m)$空间分别存取 Re*f*s(l.e)和 Slices. 标号集合 L'和 L 的空间复杂度都为 $O(m)$. 所以, 本文单子切片算法的最终空间复杂度为 $O(v \times v + v \times m)$, 与 n 无关.

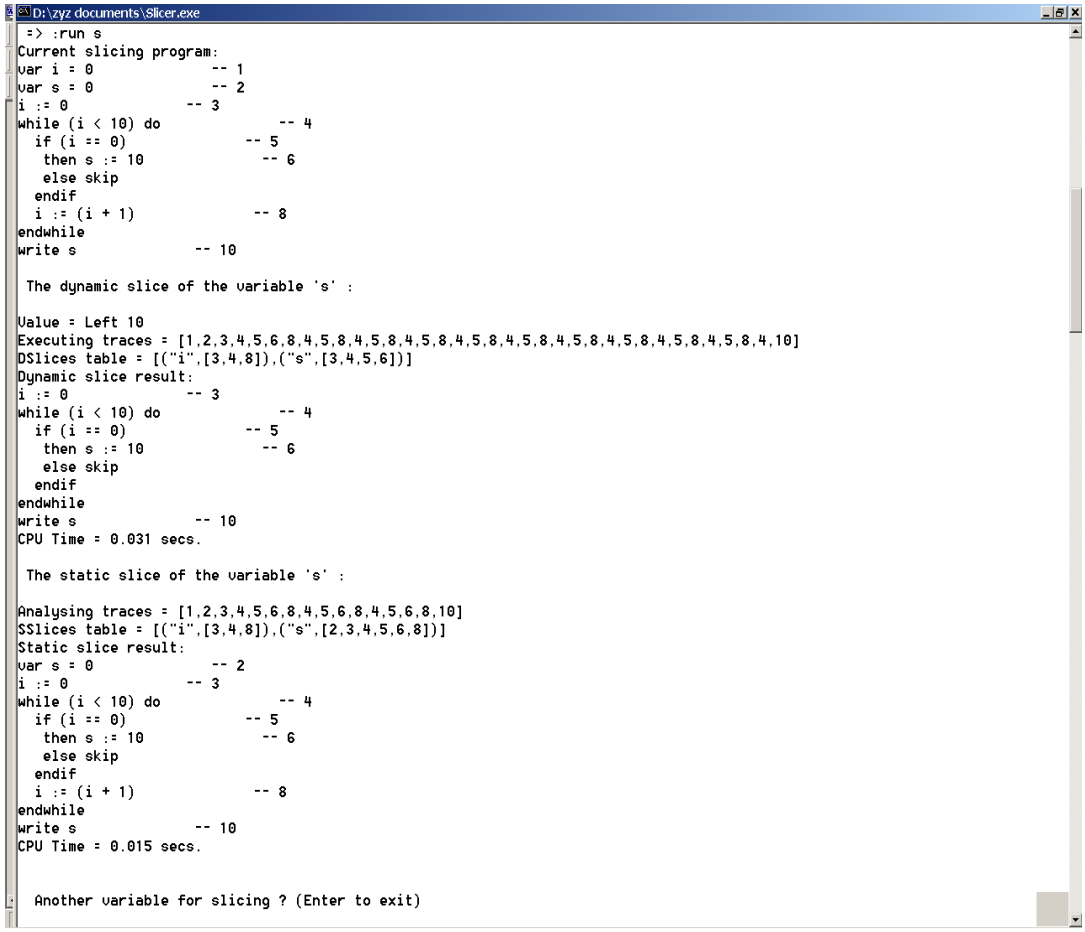

图 6　MSPS 切片器系统对图 4 程序进行动/静态切片的结果

7　实验结果分析

定理 6 曾说明了单子切片算法是可终止的. 就本文的实例语言 W 而言, 只有 While 循环语句才可能导致单子静态切片算法不终止. 在算法 1 中, 由于每次循环后, 就合并循环前后的单子切片表, 这样最终切片表必然在有限次内达到稳定, 下面利用 MSPS 系统实验验证之.

对于单层循环程序, 其循环最多次数为该循环体所有加标表达式(主要是赋值语句)的数目. 对于形如文献 [2]的特殊循环(11 个变量), 由MSPS分析路径知循环体被分析了 12 次, 最后一次是为了确认切片表的稳定.

对于循环嵌套程序, 考虑两种特殊情形. 情形 1 中每层循环体都含有 5 个变量, 且与其他嵌套层中变量都不同, 则共有 $5\times k$ 个变量, 其中 k 为总嵌套层数. 对情形 1 的 MSPS 分析路径知, 其循环次数为循环体中所有加标表达式的数目, 如从最外层循环到最里层循环的次数分别为 6, 11, 16, … , 51, 56, 61, …(各层循环都相差 5 次).

情形 2 中所有的循环体相同, 整个程序只有 5 个变量. 表 1 列出了 13 个这样多层嵌套程

序的 MSPS 实验结果, 经分析发现, 每个程序(嵌套层总数为 k)中除了嵌套最里层循环次数外, 其它层(如第 p 层, $1 \leqslant p < k$)循环次数为 $0.5\times(p+1)\times(p+2)$; 而最里层(第 k 层)循环次数 $l = 0.5\times(k+1)\times(k+2) + 3\times k$. 例如, 当嵌套层数为 12 层时, 其从外到里 12 嵌套层的循环次数分别为 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 127; 而当层数增加到 18 时, 其循环次数分别为 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 244.

表 1 多层循环嵌套程序的单子静态切片结果比较

嵌套总层数 k	6	7	8	9	10	11	12	13	14	15	16	17	18
加标表达式总数 m	40	46	52	58	64	70	76	82	88	94	100	106	112
所耗 CPU 时间 t(秒)	0.39	0.671	0.937	1.484	2.421	3.75	5.64	8.25	12.296	18.218	25.375	35.234	49.125
最里层循环次数 l	46	57	69	82	96	111	127	144	162	181	201	222	244
总分析加标表达式数 n	611	845	1133	1481	1895	2381	2945	3593	4331	5165	6101	7145	8303

关于表 1 中 13 个程序的时间消耗 t、所分析的加标表达式总数 n 与程序所有加标表达式数 m 的关系见图 7. 由情形 2 的程序结构知, m 与 k 的关系为 $m = 4 + 6 * k$.

由试验结果知, 在静态单子切片分析中, 最终分析的所有加标表达式(含重复)数目 n 与程序总加标表达式数 m 一般呈线性关系, 最坏情形下 $n = O(m^2)$.

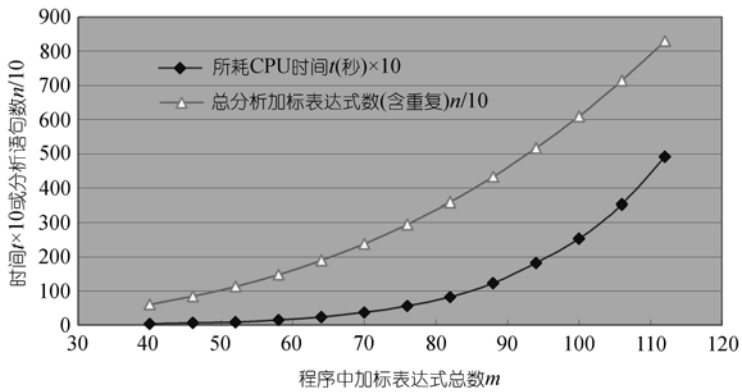

图 7 由表 1 中数据生成的关系图

所耗 CPU 时间 t、总分析加标表达式数 n 与程序总加标表达式数 m 关系图

8 相关工作

程序切片是一种重要的程序分析理解方法, 但目前切片算法还比较单一, 且是顺序增量式的, 非复合式的, 从而模块性较差. 动态切片在程序调试、错误定位等方面有着较高的应用价值, 其难点在于运行时信息的获取. 为此, 多数动态切片方法借助于关系图(如控制流图、依赖图等), 并追踪程序的执行过程, 故不可避免地需要相当大的内存空间来记录执行历史. Korel 等人 [28]曾提出了无需记录程序执行历史的正向切片方法: 将所有语句分成若干可移动的块, 当某个块执行完后由相应算法确定该块是否被包括到切片中. 这样, 在执行完最后一条语句的同时, 得到了所有语句可执行的动态切片. 但该方法不能计算潜在的依赖, 使得结果不太精确.

Hwang等人[9]曾指出程序切片算法是语言相关的, 好的切片算法应具有较强的程序语言适应性. 此外, 既然程序的行为是由其语义决定, 所以从形式化语义角度来研究程序切片是合理的.

Canfora等人的条件切片[29](conditioned slicing)将某个条件加到切片准则中, 那些不满足该条件的语句将从切片中被删去. Harman等人的无定形切片[30](amorphous slicing)允许对程序进行一些保持程序语义的简单变换. 这两种方法在求切片时适当考虑了语义的影响, 但不算真正意义上的语义切片(基于程序形式化语义的切片技术). Hausler[10]和Ouarbya等人[11]及Venkatesh[12]等研究了基于程序指称语义的切片技术, 但因传统指称语义缺乏模块性和重用性, 指称切片方法很难扩展应用到包含过程、面向对象、并发等特性的实际程序中, 即其程序语言适应性差.

针对目前程序切片方法较单一, 且其模块性和程序语言适应性较差, 我们曾在文献[16, 17]中初步探讨了一种形式化程序切片方法——模块单子切片. 在此基础上, 本文进一步完善了单子切片理论, 给出相应的性质和证明, 并重新设计切片单子转换器, 使其可直接显示地反映切片表的变化, 无需额外增加切片表的相关操作.

与现有的基于依赖图的切片方法比较, 本文算法完全从程序形式化语义角度考察程序切片, 其空间复杂度与程序执行/分析的长度(即 6.2 节中的 n)无关, 且平均每个变量单子切片的时间复杂度是线性的, 即 $O(n+m)$. 在精度方面, 由定理 3 和 4 知, 单子切片算法不会比基于依赖图的图可达性算法差, 至少能达到同等精度.

9 结论

目前程序切片方法比较单一, 主要是基于程序依赖图的可达性算法. 本文从另一个角度(即程序的形式化语义)来考察程序切片技术. 事实上, 既然程序的行为是由其语义决定的, 所以从形式化语义角度来研究程序切片是合理的.

为了增加切片算法的可重用性和语言适应性, 深入讨论了一种新的程序切片方法: 模块单子切片[16]. 利用单子转换器的特点, 将切片这类计算抽象成独立于具体语言的切片单子转换器, 若将之模块化地加载到实际程序中, 可得到相应的模块单子切片算法. 据此, 可直接在抽象语法项上计算切片, 无需在相应切片器中构造诸如控制流图或依赖图的中间结构; 在计算动态切片时也不必记录程序执行历史. 于是, 这种模块化的抽象机制使得单子切片算法具有较强的模块性和语言适应性. 此外, 还证明了模块单子切片算法的正确性和可终止性, 揭示了单子切片算法与图可达性算法间的吻合关系, 并指出其中的单子动态切片算法是精确的.

由于切片单子转换器的语言无关性和强模块性, 使单子切片方法容易扩展应用到其他更实际的程序(如含指针)[31]. 我们后续理论研究的重点是, 研究面向对象和并发程序等单子切片. 在实验方面, 我们将完善并扩展目前版本的单子切片器, 并与其他切片器产生结果进行比较分析.

致谢 作者要特别感谢南京东南大学软件工程与理论实验室对本文程序切片方法的理论支持;

感谢西班牙奥维耶多大学 J.Labra 博士与我们进行的关于单子切片算法和实现的合作交流.

参考文献

1 Tip F. A survey of program slicing techniques. J Progr Lang, 1995, 3(3): 121—189
2 Binkley D, Gallagher K B. Program slicing. Adv Comput, 1996, 43: 1—50
3 Harman M, Hierons R M. An overview of program slicing. Softw Focus, 2001, 2(3): 85—92 [DOI]
4 李必信, 郑国梁, 王云峰, 等. 一种分析和理解程序的方法——程序切片. 计算机研究与发展, 2000, 37(3): 284—291
5 李必信, 杨朝晖, 谭毅, 等. 一种基于切片技术度量 Java 耦合性的框架. 计算机学报, 2001, 24(3): 259—265
6 陈振强. 基于依赖性分析的程序切片技术研究. 博士学位论文. 南京: 东南大学, 2002: 1—5
7 Weiser M. Program slicing. IEEE Trans Softw Eng, 1984, 16(5): 498—509
8 Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment. ACM SIGPLAN Not, 1984, 19(5): 177—184 [DOI]
9 Hwang J C, Du M W, Chou C R. The influence of language semantics on program slices. In: International Conference on Computer Languages. Florida: IEEE CS Press, 1988. 120—127
10 Hausler P A. Denotational program slicing. In: 22th Annual Hawaii International Conference on System Sciences. Hawaii: IEEE CS Press, 1989. 486—495
11 Ouarbya L, Danicic S, Daoudi M, et al. A denotational interprocedural program slicer. In: Aiken P. ed. 9th IEEE Working Conference on Reverse Engineering, Virginia: IEEE CS Press, 2002. 181—189
12 Venkatesh G A. The semantic approach to program slicing. ACM SIGPLAN Not, 1991, 26(6): 107—119 [DOI]
13 Moggi E. Notions of computation and monads. Inform Comput, 1991, 93: 55—92 [DOI]
14 Mosses P D. Semantics, modularity, and rewriting logic. In: Kirchner C, Kirchner H, eds. 2nd International Workshop on Rewriting Logic and its Applications. ENTCS 15. Netherlands: Elesvier Press, 1998. 404—421
15 Zhang Y Z, Xu B W. A survey of semantic description frameworks for programming languages. ACM SIGPLAN Not, 2004, 39(3): 14—30 [DOI]
16 Zhang Y Z, Xu B W, Shi L, et al. Modular monadic program slicing. In: Yau S, Cheung P, eds. 28h Annual International Computer Software and Applications Conference, COMPSAC’04. Hong Kong: IEEE CS Press, 2004. 66—71
17 张迎周, 徐宝文. 一种基于模块单子语义的动态程序切片方法. 计算机学报, 2006, 29(4): 526—534
18 Liang S. Modular monadic semantics and compilation. Ph D Thesis. Yale: University of Yale, 1998
19 Wadler P. Comprehending monads. In: Kahn G, ed. ACM Conference on Lisp and Functional Programming. Nice: ACM Press, 1990. 61—78
20 Wadler P. The essence of functional programming. In: Sethi R, ed. 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New Mexico: ACM Press, 1992. 1—14
21 Wadler P. Monads for functional programming. In: Lecture Notes on Advanced Functional Programming Techniques. LNCS 925. Berlin: Springer-Verlag, 1995. 24—52
22 Moggi E. An abstract view of programming languages. LFCS Report, ECS-LFCS-90-113. University of Edinburgh, 1989
23 Wansbrough K. A modular monadic action semantics. Master Thesis. Auckland: University of Auckland, 1997
24 Reps T, Yang W. The semantics of program slicing and program integration. In: Diaz J, Orejas F, eds. Proceed-

ings of the Colloquium on Current Issues in Programming Languages. LNCS 352. New York: Springer-Verlag, 1989. 360—374

25 Horwitz S, Prins J, Reps T. Integrating non-interfering versions of programs. ACM Trans Program Language Sys, 1989, 11(3): 345—387 [DOI]

26 Reps T. Algebraic properties of program integration. Sci Comput Program, 1991, 17: 139—215 [DOI]

27 Kahl W. A modular interpreter built with monad transformers. Course Lectures on Functional Programming, CAS 781. 2003

28 Korel B, Yalamanchili S. Forward computation of dynamic program slices. In: Ostrand T J, ed. Proceedings of the International Symposium on Software Testing and Analysis. Washington: ACM Press, 1994. 66—79

29 Canfora G, Cimitile A, De Lucia A. Conditioned program slicing. Inform Software Tech, 1998, 40(11/12): 595—607 [DOI]

30 Harman M, Binkley D, Danicic S. Amorphous program slicing. J Syst Software, 2003, 68(1): 45—64 [DOI]

31 Wu Z Q, Zhang Y Z, Xu B W. Modular monadic slicing in the presence of pointers. In: Alexandrov V N, Albada G D, Sloot P M, eds. 6th International Conference on Computational Science. LNCS 3994. Reading: Springer-Verlag, 2006. 748—756