含指针程序的单子切片方法

张迎周[1),2),3)] 吴重强[4)] 钱 巨[2),5)] 张卫丰[1),2)] 徐宝文[2),6)]

[1)](南京邮电大学计算机学院 南京 210003)

[2)](南京大学计算机软件新技术国家重点实验室 南京 210093)

[3)](北京邮电大学网络与交换技术国家重点实验室 北京 100876)

[4)](EMC中国研发中心 上海 200433)

[5)](南京航空航天大学信息科学与技术学院 南京 210016)

[6)](南京大学计算机科学与技术系 南京 210093)

摘 要 传统的含指针程序切片方法将指向分析与切片计算分开,增加了一定系统开销,为此文中提出一种可同时进行切片计算和指向分析的单子切片算法.该算法将程序正向切片思想与数据流迭代分析相结合,它是流敏感的,具有一定的精度,而且因指向分析和切片计算同时进行,故不需要像一般的流敏感分析方法那样记录每一个程序点的指向信息,而只需记录当前所分析的程序点处指向信息,从而节省了存储空间.此外,它还继承了原有单子切片方法所具有的强语言适应性和组合性.

关键词 程序切片;单子切片;数据流迭代;指针;指向分析

中图法分类号 TP311 **DOI号**: 10.3724/SP.J.1016.2010.00473

A Monadic Slicing Algorithm for a Program with Pointers

ZHANG Ying-Zhou[1),2),3)] WU Zhong-Qiang[4)] QIAN Ju[2),5)] ZHANG Wei-Feng[1),2)] XU Bao-Wen[2),6)]

[1)](*College of Computer, Nanjing University of Posts and Telecommunications, Nanjing* 210003)

[2)](*State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing* 210093)

[3)](*State key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing* 100876)

[4)](*EMC China R&D Center, Shanghai* 200433)

[5)](*College of Information Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing* 210016)

[6)](*Department of Computer Science and Technology, Nanjing University, Nanjing* 210093)

Abstract Program slicing is a family of program decomposition techniques. By introducing point-to analysis to the previous monadic slicing, the authors present and implement an approach of monadic slicing for a program with pointers. This approach obtains the point-to information through the data-flow iteration. Being different from the traditional methods, the point-to information and slicing are computed in the same phase in the method, by combining the forward monad slicing with data-flow iteration. Instead of recording point-to information for every statement, it is only needed to record the information for current analysis statements. So the method saves space without losing the precision. In addition, the approach also reserve the excellent properties of compositionality and language-flexibility from the original monadic slicing method.

Keywords program slicing; monadic slicing; data-flow iteration; pointer; point-to analysis

---

收稿日期:2009-04-13;最终修改稿收到日期:2009-08-06. 本课题得到国家自然科学基金(60703086,90818027,60633010,60873049,60973046,60903026)、国家"八六三"高技术研究发展计划目标导向类项目(2009AA01Z147)、国家"九七三"重点基础研究发展规划项目基金(2009CB320703)资助.**张迎周**,男,1978年生,博士,副教授,研究方向为软件形式化方法、程序切片等. E-mail: zhangyz@njupt.edu.cn. **吴重强**,男,1983年生,硕士研究生,研究方向为程序分析、软件形式化方法. **钱 巨**,男,1981年生,博士,研究方向为程序分析、软件测试. **张卫丰**,男,1975年生,博士,研究方向为软件测试、信息检索. **徐宝文**,男,1961年生,博士,教授,博士生导师,研究领域为程序设计语言、软件形式化技术和程序分析等.

1 引 言

程序切片是一种程序分解技术,它通过从源程序中抽取与某项计算相关的语句,得到一个规模较小的程序,可体现源程序行为的一个子集,这使我们的精力得以集中到程序中的一小部分[1].由于切片技术可以将复杂的工作简单化甚至自动化,它被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[2-6].

为丰富现有的程序切片算法,我们曾从形式化语义的角度提出一种新型切片方法——模块单子切片[7-10],它基于程序的模块单子语义[11-13].单子切片算法具有较强的可扩展性和可重用性,支持组合式程序设计模式,反映现代程序设计语言(如C++、Java等)中模块化设计特性.本文在此工作基础上,完善并扩展文献[9-10]中的单子切片方法,使之能求含指针程序的单子切片,并以此展示我们单子切片方法的易扩展性.

指针的出现可导致别名问题[14-16](即多个变量访问同一内存位置),故需要进行指针分析以获得相应的数据依赖信息.指针的引入给程序分析带来了3个主要问题:(1)允许动态申请内存后,可以构造像链表这样具有静态不确定长度的递归数据结构,由此存在如何用有限的方式表示它的问题;(2)指针的引入需更新(泛化)数据依赖在新框架下的定义;(3)由于静态分析时指针指向的不确定性,需要考虑如何做指向分析,以确定变量的指向集.第1个问题的解决将提供一种表示程序中变量的有效方式,有助于定义新框架下的数据依赖和指向分析,是解决后两个问题的基础.第2个问题的解决将给出新框架下的数据依赖定义,从概念上包容了引入指针后的新情况,而该定义的具体实现则依赖于指向分析.也就是说,第1个问题是基础,第2个问题提出了概念上的目标,而第3个问题的解决方案则帮助实现该目标.

在已有工作[7,9]的基础上,本文将考虑含指针的模块单子切片方法.本文重点处理的是第3个问题(关于指向分析的),对于前两个问题采用已有的简单方法处理.本文的切片方法利用原有单子切片中正向切片[17]的思想,使用数据流迭代进行指向分析.这种分析方法有一定的精度,且与正向单子切片方法结合后,指向分析和切片计算可同时进行.

本文先简介一个实例语言的单子语义及其相应的单子静态切片算法;然后分析研究指针引入后的问题及其解决方法,并据此给出扩展指针后的单子切片算法以及一个具体的求含指针程序单子切片实例;最后实现所提出的指针单子切片算法,并分析其相应的时空复杂度.

2 预备知识

单子概念最初是在20世纪50年代作为范畴论里一种函子而被提出的.在1989年由Moggi将之引入到语义框架中.一般地,单子可形式化地表示成三元组$(m, return_m, bind_m)$,其中m是类型构子,$return_m$和$bind_m$是其两个基本操作,且满足3个规律(即左幺元、右幺元和结合律,详见文献[18]).

在利用单子描述程序语义时,常使用单子转换器来组合多个单子.单子转换器由类型构造器t及其提升函数$lift$构成,其中t将某个给定单子$(m, return_m, bind_m)$映射到新单子$(t\ m, return_{t\ m}, bind_{t\ m})$.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.

因单子转换器完全独立于具体的语言,仅表示某类计算,所以单子转换器的设计是有意义的.目前人们已设计了不少单子转换器,如统一描述与程序环境交互计算的环境单子转换器EnvT,表示与状态相关计算的状态单子转换器StateT等[11].在文献[9]中,我们给出了如下的切片单子转换器SliceT,以此来统一描述程序切片这一类计算.SliceT定义中L表示进行当前计算所需表达式的标号集合;$rdLabels$,$inLabels$和$getSli$,$setSli$分别是SliceT中参数L和s(切片表)的读取和更新操作.

$$\textbf{type}\ \mathrm{SliceT}\ L\ s\ m\ a = (L, s) \rightarrow m(a, s)$$

$$return_{\mathrm{SliceT}\ L\ s\ m}\, x = \lambda(L, s).return_m(x, s)$$

$$e\ `bind_{\mathrm{SliceT}\ L\ s\ m}'\ f = \lambda(L, s).\{(a, s') \leftarrow e(L, s);\ fa(L, s')\}_m$$

$$lift_{\mathrm{SliceT}\ L\ s}\, e = \lambda(L, s).\{a \leftarrow e;\ return_m(a, s)\}_m$$

$$rdLabels = \lambda(L, _).return_m(L, ())$$

$$inLabels\ L\ c = \lambda(L', s).c(L, s)$$

$$getSli = \lambda(_, s).return_m(s, s)$$

$$setSli\ s = \lambda(_, s').return_m((), s)$$

文献[9]中,我们还详细证明了单子切片算法的正确性和可终止性以及其与基于依赖图切片算法间的吻合性.

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，单子 m 可被分解成一系列的单子转换器，每个代表一种计算.换句话说，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.

为讨论方便，本文仍考虑文献[9]中的简单命令式实例语言 W，其抽象语法如下(后面将对其进行含指针程序的扩展，详见第 4 节).

论域：ide：Ide(标识符)；l：Label(标号)；

e：Exp(表达式)

抽象语法：

$S ::= ide := l.e \mid S_1 ; S_2 \mid \text{skip} \mid \text{read } l.ide \mid \text{write } l.e$

$\mid \text{if } l.e \text{ then } S_1 \text{ else } S_2 \text{ endif} \mid \text{while } l.e \text{ do } S$

endwhile.

也假设上述表达式 e 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，且该标号是针对整个表达式的，其子表达式不再有标号.为后面计算切片方便，对 Read 语句中变量也赋予一个标号.

为了从切片表中标号集合 L 获得一个符合 W 文法的程序切片，我们曾给出了 W 语言的 $Syn(s,L)$ 函数定义[7,9]，其中 s 表示被分析的 W 源程序. $Syn(s,L)$ 说明如何根据所求得的切片标号集合 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.更主要地，它允许我们只需关注所分析程序中的加标表达式，这是因为：程序切片的主体部分依赖于语句中的表达式，其它部分可由 $Syn(s,L)$ 捕获.

3　模块单子静态切片算法

通过将程序切片这类计算抽象成切片单子转换器 SliceT[7,9-10]，其它的单子(如状态单子 StateMonad 和环境单子 EvnMonad[12-13])可很容易地被转换成切片单子 SliceMonad[9-10].同样地，其它单子转换器 t 也可将 SliceMonad 转换成其它相应的单子.由此，可方便将切片计算融合入已有的单子语义描述中.

文献[7,9]中详细叙述了过程内程序的单子切片(包括静态切片和动态切片)技术.单子静态切片算法考虑程序最终点的单变量程序切片，对应切片标准为 $\langle p, v\rangle$，其中 p 为程序最后语句点，v 为程序中某个变量.算法基本思想为：先将切片单子转换器组合到语义模块描述中，如 ComptM≡(SliceT · StateT · EnvT) IO，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并计算相应的静态切片，最后得到所要求的切片.

算法中的切片数据结构为 Slices＝[(Var, $Labels$)]，它是由变量及其切片(标号集合 $Labels$＝[Int])所构成的一张表(Hash 表)，并包含 3 个基本操作函数：$lkpSli$、$updSli$ 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.此外，算法中需要按下式计算加标表达式的 L'，以此来捕获相应的控制依赖和数据依赖[7,9]：

$$L' = \{l\} \cup L \cup \bigcup_{r \in Refs(l.e)} lkpSli(r, getSli) \quad (1)$$

其中 $Refs(l.e)$ 表示所有出现在表达式 $l.e$ 中变量的集合，即 $l.e$ 的引用集.

因静态切片中不需关心值的计算结果，故静态切片语义描述中省略了表达式值的计算语义，具体的 W 语言静态单子切片语义描述如下(其中符号 Fix 表示不动点算子).

$[\![ide := l.e]\!] = \{L \leftarrow rdLabels;\ L' \leftarrow \{l\} \cup L \cup$
$\bigcup_{r \in Refs(l.e)} lkpSli(r, getSli);\ updSli(ide, L', getSli)\}$;

$[\![c_1 ; c_2]\!] = \{[\![c_1]\!]; [\![c_2]\!]\}$;

$[\![\text{skip}]\!] = return()$;

$[\![\text{if } l.e \text{ then } c_1 \text{ else } c_2 \text{ endif}]\!] = \{L \leftarrow rdLabels;$
$L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in Refs(l.e)} lkpSli(r, getSli);$
$T \leftarrow getSli; inLabels\ L' [\![c_1]\!]; T1 \leftarrow getSli; setSli(T);$
$inLabels\ L' [\![c_2]\!];\ T2 \leftarrow getSli; mrgSli(T1, T2)\}$;

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

$[\![read\ l.ide]\!] = \{L \leftarrow rdLabels; L' \leftarrow \{l\} \cup L;$
$updSli(ide, L', getSli)\}$;

$[\![\text{write } l.e]\!] = return()$.

由此，可得到最终静态切片表 Slices，它包含了所分析程序中所有单变量的静态切片.

通过与西班牙奥维耶多大学 Labra 博士合作[19]，我们开发了一个过程内单子切片器原型系统.该原型以程序的模块单子语义为基础，支持增量式开发，其实现语言为 Haskell[20-21].

下节及后面将据过程内单子静态切片算法扩展讨论含指针的单子切片方法.

4　含指针程序的单子切片算法

4.1　引入指针后的问题分析

本文引言中已指出，程序中指针的引入，将给数

据流分析带来 3 个主要问题. 对于第 1 个问题,即用有限方式表示递归数据结构的问题,许多学者已提出了解决方法[22-24]. 这些方法基本上都是利用某种近似将无限的结构有限化,差别在于近似的程度不同. 较粗糙的方法是将系统的堆空间看作一个整体来处理,更精确的做法需采用较为高级的形分析(shape analysis)技术[15,25-26]. Chase 等人[24]利用指针变量区分堆空间中的变量,并为每个程序点求取存储状态图(Storage Shape Graph),其结果尽管更加精确,但复杂度较高,且所采用的概要节点(summary node)仍是一种近似.

本节拟采用的方法,在一定程度上区分了堆空间中的变量,但出于算法复杂度的考虑,并没有达到像 Chase 那样的精确程度. 我们认为所有在同一程序点处申请的堆空间组成了一个数组,并将这个数组当成整体处理. 由于本节旨在将指针融入我们的模块单子切片算法中,故采用了这种简单而又具一定精确度的方法.

第 2 个问题(关于更新数据依赖定义的)与指针引入后产生的别名问题有一定关系. 由于第 1 个问题的解决保证了对程序中变量的有限表示,因此可以采用文献[27]中方法,依据抽象内存地址的可能定义和使用来泛化新框架下的数据依赖. 在我们的指针单子切片算法中,我们将通过指向分析、扩展赋值语句的语法及重新定义引用集 $Refs(l.e)$ 来实现泛化. 此外,我们也将考虑赋值语句中左值表达式对变量的可能引用情况,详见后面的具体算法.

最后一个问题是关于指针分析的,这是本文后面研究的重点. 指针分析的最终目的是要获取别名信息. 有两种表示别名信息的方法:别名对(alias pair)表示法[14-15]和指向(point-to)表示法[16],分别对应于别名分析和指向分析. 指向分析相对别名分析获得的结果更紧凑,且易于后继分析的使用,故本文采用指向分析. 根据分析时是否采用控制流信息,指向分析可分为流敏感[14,25]和流不敏感[28-30]两种. 流敏感的分析,考虑控制流,需要迭代,故效率相对较低,但精确度较高,可以得到每一程序点的指向信息. 而流不敏感的分析,不考虑控制流,认为程序中的语句可以按任意次序执行,因此它只能为每个变量全局地(或在一定范围内)确定一个指向集,而不能精确到程序点. 本文的指针分析算法是种流敏感方法.

如果考虑过程间指针情况,可按是否考虑上下文的影响将指针分析算法分为:上下文敏感指针分析[16,31]和上下文不敏感指针分析[25]. 上下文不敏感的指针分析算法中,如果一个函数被多次调用,则此函数每个调用点的指针别名信息将被合并在一起,然后用合并后的指针别名信息对该函数指针分析,并把结果传递到每个调用点之后处. 上下文敏感的过程间指针分析方法可按一定的准则对同一函数不同调用点的调用上下文进行区别,进而对不同的调用上下文产生不同的指针分析结果. 本文还主要对过程内的指针进行分析,暂不涉及上下文是否敏感的问题.

我们曾提出的单子切片算法采用了正向切片思想,可适于数据流迭代方法的嵌入,因此本节将采用数据流迭代方法来分析指向.

4.2 算 法

结合我们单子切片方法和数据流迭代的指向分析,本节将给出含指针的过程内静态切片算法. 本节仍以本文开始所给的 W 语言为例,待解决的关键问题是对赋值语句的处理,其它的语句(如条件语句、循环语句等)只要在原有单子切片框架下加入指向分析的计算即可. 为使 W 语言包含指针,我们将其赋值语句增加左值表达式,并对之进行处理. 以下,我们将首先给出指向集数据结构,然后讨论含左值表达式赋值语句的形式及其处理方法,最后提及其它语句的处理.

为进行指向分析,我们设计如下的指向集数据结构:

```
type Var=String
type PtSet=[Var]
type PT=[(Var, PtSet)]
    getPT :: ComptM PT;
    setPT :: PT→ComptM PT;
    lkpPT :: Var→PT→ComptM PtSet;
    updPT :: Var→PtSet→PT→ComptM();
    mrgPT :: PT→PT→ComptM PT;
```

每个变量所对应的指向集是一个变量集合,所有变量及其指向集构成了一张表(Hash 表),记为指向集类型 PT. 它包括 5 个基本的操作函数:*getPT*、*setPT*、*lkpPT*、*updPT* 和 *mrgPT*,分别用来获取和设置当前指向集表 PT、查找 PT 中某变量对应的指向集(变量集合)、更新 PT 中数据和合并两个 PT.

为后面讨论的方便,引入强更新(strong update)和弱更新(weak update)的概念. 在下面扩展算法中,经常需要更新变量的切片信息与指向信息. 如果先将变量的原有信息删去,再将新信息赋给变量,称为强更新,是一种替换更新操作,如前面的

updSli 和 *updPT* 操作函数. 如果是在保留变量原有信息的基础上，将新信息添加入变量信息中，称为弱更新，即因存在可能指向关系，从而保守地将相应信息合并来更新. 切片信息和指向信息的弱更新操作函数分别为下面新引入的 *xtdSli* 和 *xtdPT*：

$xtdPT$ :: [Var] → PtSet → PT → ComptM()

$xtdSli$ :: [Var] → Labels → Slices → ComptM()

引入指针后可能需要同时更新多个变量的切片信息和指向信息，故 *xtdPT* 和 *xtdSli* 操作函数的第 1 个参数是个变量列表. 为了更大程度与引入指针前单子切片算法兼容，这里没有像我们文献[10,46]那样修改 *updSli*，而是增加了 *xtdSli* 函数.

算法 1.　表达式的引用集和指向集算法.

输入：表达式 e

输出：e 的引用集 *Refs* 和指向集 *PtInfo*

算法过程：

```
case e of
  e 是形如 &y 的表达式 →
      Refs(e)=∅;   PtInfo(e)={y};
  e 是形如 *y 的表达式 →
      令 y 的当前指向集 ys=lkpPT(y,getPT);
      Refs(e)={y}∪ys;
      PtInfo(e)= ∪_{v∈ys} lkpPT(v,getPT);
  e 是单纯变量 y →
      令 y 的当前指向集 ys=lkpPT(y,getPT);
      Refs(e)={y}; PtInfo(e)=ys;
  e 是由某操作符连接两个子表达式 e1 和 e2 构成的表达式 →
      Refs(e)=Refs(e1)∪Refs(e2);
      PtInfo(e)=PtInfo(e1)∪PtInfo(e2);
  e 是数值、字符、逻辑等类型常量 →
      Refs(e)=∅;   PtInfo(e)=∅;
end case.
```

下面考虑如何将指针扩展到现有 W 语言中，并进行相应切片计算和指向分析.

先对 W 语言赋值语句 $ide := l.e$ 进行指针扩展. 在仅考虑单级解引用的情况下(多级解引用可分解为多个单级解引用处理)，$ide := l.e$ 的指针扩展主要包括两方面：(1) *ide* 扩展包含形如 $*x$ 的左值表达式符号；(2) $l.e$ 扩展包含形如 $*y$ 和 $\&y$ 的右值表达式. 此外，形如 $x := l.\&y$ 的赋值语句可用于表示申请堆空间的语句(y 被看成一个数组来整体处理). 关于数组，我们采用类似文献[32]中保守方法，也把数组看作一个整体来处理，即对数组中某个元素的任何更新被看成是对整个数组的更新和引用. 这样，扩展后的表达式 $l.e$ 中出现的变量可分为 3 类：(1) 被引用变量；(2) 被解引用变量；(3) 被取地址变量. 从而，引用集 $Refs(l.e)$ 除了包括前两类变量外，还包括第 2 类变量被解引用后可能引用到的变量，即

$$Refs(l.e)= \{x \mid x \text{ 是被引用变量}\} \cup \{y \mid y \text{ 是被解引用变量}\} \cup \{z \mid z \in lkpPT(y, getPT), y \text{ 是被解引用变量}\}.$$

详细的 $Refs(l.e)$ 计算见算法 1. 算法 1 中 $PtInfo(l.e)$ 函数是为进行指向分析而引入的，可计算表达式 $l.e$ 所可能产生的指向集. 例如，对于右值表达式 y 产生的指向集就是变量 y 的指向集，即 $lkpPT(y, getPT)$；而对于右值表达式 $*y$ 产生的指向集是变量 y 指向集中所有变量指向集的并集，即

$$\bigcup_{v \in lkpPT(y, getPT)} lkpPT(v, getPT).$$

通过算法 1 可解决右值表达式的变量引用和指向情况，可允许我们对赋值语句进行统一处理，而不必像文献[10,46]那样分 8 种情况分别处理，且使其处理情况更具有一般性(如可处理诸如 $*x+3$ 含指针运算的表达式)，从而更方便扩展现有的单子切片算法求含指针的 W 语言程序切片和指向信息.

对于扩展指针后的赋值语句，根据左值表达式是否为被解引用变量来强更新和弱更新相应变量的切片和指向集，其具体的包含切片计算和指向分析的单子静态切片语义描述为

```
[[ide := l.e]] =
{据 rdLabels, getSli 和 getPT 分别获取当前标号集合 L, 切片表 T 和指向集表 P;
 由算法 1 计算表达式 l.e 的引用集 rs=Refs(l.e), 指向集 ps=PtInfo(l.e);
 if (ide 是形如 *x 的左值表达式符号) then
   由式(1)计算临时标号集合
       L′={l}∪L ∪ ∪_{r∈rs∪{x}} lkpSli(r,T);
   令变量 x 的当前指向关系集 xs=lkpPT(x,P);
   以 L′ 弱更新(合并更新) xs 中所有变量切片, 即 xtdSli(xs,L′,T);
   以 ps 弱更新 xs 中所有变量指向关系集, 即 xtdPT(xs,ps,P);
 else
   由式(1)计算临时标号集合
       L′={l}∪L ∪ ∪_{r∈rs} lkpSli(r,T);
   以 L′ 强更新(替换更新) ide 的切片, 即 updSli(ide,L′,T);
   以 ps 强更新 ide 的指向关系集, 即 updPT(ide,ps,P);
 end if.
}
```

至于 W 语言的其它语句在扩展指针情况后,由于正向单子切片思想与数据流迭代方法的相似性,所以只需要在原有切片算法中加入相关指向分析计算即可.具体地,扩展指针后的 W 程序静态单子切片语义描述为

$[\![ide := l.e]\!] = \{L \leftarrow rdLabels;\ T \leftarrow getSli;\ P \leftarrow getPT;$

$ps = PtInfo(l.e);\ xs = lkpPT(x, P);$

if (ide 形如 $*x$) then

$L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in Refs(l.e) \cup \{x\}} lkpSli(r, T);$

$xtdSli(xs, L', T);\ xtdPT(xs, ps, P);$

else

$L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in Refs(l.e)} lkpSli(r, getSli);$

$updSli(ide, L', T); updPT(ide, ps, P)\};$

$[\![c_1 ; c_2]\!] = \{[\![c_1]\!]; [\![c_2]\!]\};$

$[\![\text{skip}]\!] = return();$

$[\![\text{if } l.e \text{ then } c_1 \text{ else } c_2 \text{ endif}]\!] = \{L \leftarrow rdLabels;$

$L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in Refs(l.e)} lkpSli(r, getSli);$

$T \leftarrow getSli; P \leftarrow getPT; inLabels\ L' [\![c_1]\!];$

$T1 \leftarrow getSli; P1 \leftarrow getPT; setSli(T); setPT(P);$

$inLabels\ L' [\![c_2]\!]; T2 \leftarrow getSli; P2 \leftarrow getPT;$

$mrgSli(T1, T2); mrgPT(P1, P2)\};$

$[\![\text{while } l.e \text{ do } c \text{ endwhile}]\!] = Fix(\lambda f.\{L \leftarrow rdLabels;$

$L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in Refs(l.e)} lkpSli(r, getSli);$

$T \leftarrow getSli; P \leftarrow getPT;$

$f \bullet \{inLabels\ L' [\![c]\!]; T' \leftarrow getSli; P' \leftarrow getPT;$

$mrgSli(T, T');\ mrgPT(P, P')\}\});$

$[\![\text{read } l.ide]\!] = \{L \leftarrow rdLabels;\ L' \leftarrow \{l\} \cup L;$

if(ide 形如 $*x$) then

$xtdSli(lkpPT(x, getPT), L', getSli)$

else $updSli(ide, L', getSli)\};$

$[\![\text{write } l.e]\!] = return().$

与第 3 节切片语义描述相比较,引入指针后的单子静态切片语义描述(算法)是在原有算法基础上较独立地增加指向分析代码而成,可参见上述描述中加亮部分的代码.因算法 1 统一处理了引入指针后表达式的变量引用和指向情况,从而避免了文献[10,46]算法中显现地处理 8 种类型赋值语句,使得我们最终的算法描述更简洁,更能体现了我们单子切片算法的语言强适应性和可组合性.此外,我们算法是流敏感的,具有一定的精度,而且因指向分析和切片计算同时进行,故不需记录每一个程序点的指向信息,而只需记录当前程序点的指向信息.即随着程序分析的进行,我们不断更新切片表和指向信息表,而不保留中间分析过程中的指向信息,从而节省了存储空间.

过程内单子切片算法已被证明是可终止的[9].此外,在扩展指针后的单子切片算法中,将所有在同一程序点处申请的堆空间看成一数组,并作为整体处理,以此提供一种表示程序中变量的有限方式,从而保证了本文算法的切片计算和指向分析的可行性和可终止性.

5 实例分析

为进一步说明本节的指针单子切片算法,下面分析一个具体的程序(见图 1),其表达式的标号为该表达式所在语句片断的标号.在以下讨论中,我们只给出切片表和指向集变化部分的信息,并令 $L(v)$ 表示变量 v 的切片,$P(v)$ 表示变量 v 的指向集,其初始值均为空.

由上节切片语义中赋值语句的算法描述知,在第 1 号表达式被分析后,其切片表中变量 $flag$ 的切片被更新为:$L(flag)=\{1\}$.第 2～4 号语句组成了一个条件语句,它给变量 s 赋地址值.由指针切片算法(第 4 节切片语义描述)中条件语句和赋值语句部分描述知,在该条件语句分析过后,变量 s 的切片和指向集分别被更新为:$L(s)=\{1,2,3,4\}$;$P(s)=\{a,b\}$.类似地,经过 5 号表达式后,变量 c 的切片更新为:$L(c)=\{5\}$.

第 6～9 号语句组成了一个循环语句,按第 4 节切片语义描述算法需要迭代直至所有变量的切片及指向集达到稳定.在第一次循环后,变量 $flag$,t,a,b 的相关信息发生了变化:

$L(flag)=\{1,6,9\};$

$L(t)=\{1,6,7\};$

$L(a)=\{1,2,3,4,5,6,7,8\};$

$L(b)=\{1,2,3,4,5,6,7,8\};$

$P(t)=\{c\}.$

进行第 2 次循环后,变量 a,b,t 的切片信息变化为

$L(a)=\{1,2,3,4,5,6,7,8,9\};$

$L(b)=\{1,2,3,4,5,6,7,8,9\};$

$L(t)=\{1,6,7,9\}.$

因切片表和指向集不稳定,需进入第 3 次循环,该次循环分析后,均达到了稳定,退出循环语句.

经 10 号语句后,变量信息不发生任何变化,其最终的切片结果和指向集情况见表 1,表中 $L(Var)$ 和 $P(Var)$ 分别表示变量 Var 的切片和指向集.

表 1　最终的切片和指向集结果

Var	*L*(*Var*)	*P*(*Var*)
flag	{1,6,9}	∅
s	{1,2,3,4}	{*a*,*b*}
c	{5}	∅
t	{1,6,7,9}	{*c*}
a	{1,2,3,4,5,6,7,8,9}	∅
b	{1,2,3,4,5,6,7,8,9}	∅

显然，可由以上的切片信息和 $Syn(s,L)$[9] 获得变量的最终切片结果，即所分析程序的相应片断. 例如：图 1 程序关于〈10,*a*〉的最终静态切片是{1,2,3,4,5,6,7,9,10}，其中语句 10 是根据 $Syn(s,L)$ 中 write 规则加入的.

```
1.    flag := 1;
2.    if flag<5 then
3.      s := &a
4.    else s := &b
      endif;
5.    c := 1;
6.    while flag<5 do
7.      t := &c;
8.       *s := *t;
9.      flag := flag+1
      endwhile;
10.   write a
```

图 1　一个含指针的实例程序

6　算法实现及复杂性分析

本文的含指针单子切片算法是在我们之前实现的过程内单子切片器[9,19]基础上实现的，根据第 4 节切片语义描述(尤其是加粗部分)算法，可方便地将指针扩展到现有的单子切片器中.

此外，随着程序规模的加大，切片表也随之扩大，为了降低切片表中标号集合操作的时间和空间复杂度，本文采用 Haskell 自带的函数库 IntSet. 模块 IntSet 中函数算法是基于 Patricia 树结构的[33]，其大部分操作(如插入、删除)最坏情况下时间复杂度为 $O(\min(s,C))$，其中 s 为集合元素个数，C 为与机器相关的常数 32 或 64；集合的交、并、差的最坏时间复杂度为 $O(s)$.

由我们最终开发出来的单子切片器对图 1 程序切片分析结果见图 2. 结果包含了分析路径、切片表、指向集表、关于〈10,*a*〉的切片结果以及最终所耗的 CPU 时间，运行机器基本配置为：Intel Pentium IV 2.93GHz 处理器、512MB 内存.

扩展指针后的单子切片算法描述在过程内单子静态切片算法基础上增加了指向分析，结合算法 1 分析知，指向分析的最坏时间复杂度要小于切片计算的最坏时间复杂度，所以只考虑切片计算的时间复杂度. 由于模块单子编译器/解释器可自动、安全地将我们的切片单子转换器模块化加载到语义模块中，所以单子切片算法中语句的切片时间复杂度分析主要是针对具体程序语言的 L'. 本文所定义的 L' 时间消耗主要包括查找切片表(Hash 表)和合并相应标号集合(Patricia 树)，其时间复杂度分别为 $O(v)$ 和 $O(v\times m)$，其中 v 为程序中单变量数目，m 为程序中加标表达式的数目，故 L' 的最坏时间复杂度为 $O(v\times m)$. 于是过程内单子静态切片算法的最坏时间复杂度为 $O(v\times m\times n)$，其中 n 为程序中实际被分析的加标表达式(含重复)数目. 类似地，由文献[9]终止性分析知，n 与 m 一般呈线性关系，最坏情况下 $n=O(m^2)$.

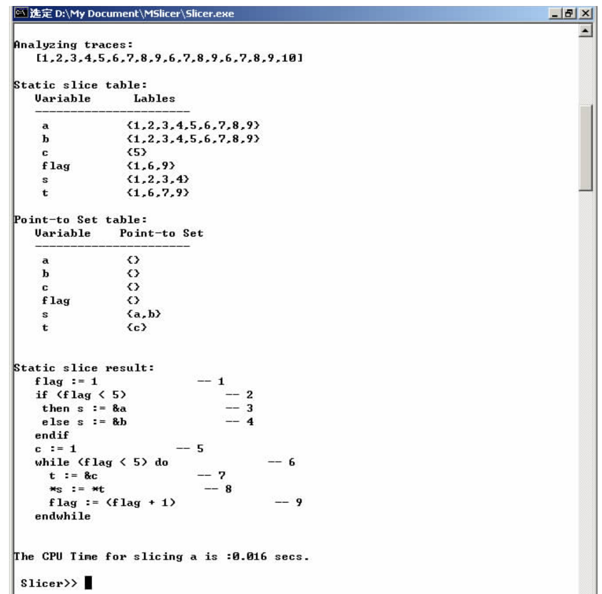

图 2　单子切片器对图 1 程序进行切片结果

对于指针扩展算法的切片计算，可能要同时更新 v 个变量的切片，一次更新需耗时 $O(v)$，最坏情况下，新增切片计算总耗时 $O(v\times m)$，这与上述 L' 的最坏时间复杂度一致，故扩展指针后算法并未增加原有算法的时间复杂度.

由文献[9]复杂度分析知，过程内单子静态切片算法的空间复杂度为 $O(v\times m+v\times v)$. 对于扩展指针后单子切片算法的空间复杂度，主要是增加指向集情况. 指向集表空间耗费为 $O(v\times v)$，故也未增加原有算法的空间复杂度.

7　相关工作

目前关于指针分析综述性的文章有文献[34]和

文献[35]等. 文献[34]比较了5种C语言指针分析方法的精度和效率,讨论了它们对后续分析的影响. 文献[35]则总结了目前指针分析的相关研究成果及尚待解决的问题.

指针分析是一个不可判定问题[36],相关的近似算法都在效率和精度间权衡. 显然,流敏感和上下文敏感的指针分析算法的精度较高,但效率较低. 故Hind等人[34,37]建议,我们应根据实际情况的需要来选择指针分析算法,并注意到不同分析方法间的影响. 他们指出,在上下文敏感的指针分析中,流敏感分析在精度上对基于子集的流不敏感分析提高不大. 类似地,Ruf[38]指出上下文敏感分析并不能提高通常流敏感分析的精度.

为了在含指针程序切片中考虑可能别名,Horwitz等人[27]提出另一种泛化数据依赖概念的方法,该方法是基于抽象内存表示的可能定义和使用来泛化数据依赖. 与该思想类似,Agrawal等人[39]给出了一种可同时处理数组和指针的基于PDG静态切片算法. 将指针引入程序切片后,除了要泛化数据依赖外,可达定义也要作相应的改变以便能考虑左值表达式. Tip[2]根据左值表达式所对应的内存地址布局给出了新的可达定义.

Jiang等人[40]给出了含指针和数组的C程序切片算法,为指针的解引用和操作地址增加哑元变量和标识符,将指针引用和赋值看成是对相应哑元变量的使用和修改. Tip在文献[2]中示例指出该算法有缺陷. Lyle等人[32]给指针变量赋地址,并对语句中每个指针的所有可能地址构建成表. Ernst[41]给出了一个指针切片器,将存储显示地表示成聚集值(aggregate value)来支持指针操作. 为了处理多级指针的间接引用和赋值问题,Lyle等人[42]构建了指针状态图(Pointer State Graph,PSS). Ross等人[43]提出了一种约简的PDG来处理可能别名问题.

不同于将指向分析与切片计算分开的上述切片算法,本文算法将正向程序切片思想与数据流迭代分析相结合,可同时进行切片计算和指向分析,从而比一般的数据流迭代算法节省空间,而且其流敏感分析保证了一定的精度. 此外,它还继承了原有单子切片方法所具有的强扩展性和程序语言适应性的优点. 本文扩展指针后算法较扩展前单子切片算法时空复杂度都没有增加,且也经一次计算就可得到所有变量的切片. 另外,本文因利用Haskell实现切片算法,Haskell的惰性计算(lazy computing)特性,使得本文的算法效率更高,即只是在需要的时候才计算相应的切片表和指向集等.

至于其它语义切片方面,Ouarbya等人[44]虽在十年后将Hausler[45]提出的指称切片(基于程序指称语义的切片方法)从过程内扩展到过程间,但目前未见进一步扩展(如扩展包含指针等). 这主要原因是传统指称语义缺乏模块性和重用性,指称切片方法的程序语言适应性较差,阻碍了其进一步扩展.

我们在文献[7,9-10]中详细地给出了单子静态切片理论和实现方法,并在文献[10,46]中曾初步探讨了有关含指针程序的单子切片算法. 比较而言,本文进一步完善文献[10,46]中指向分析相关概念(包括强/弱更新等)、指向集数据结构和算法的复杂度分析;给出求表达式引用集和指向集的算法(算法1);重新设计指针单子切片算法,对赋值语句进行统一处理,不再分成8种情况分别处理,使得最终算法更能展示单子切片方法的易扩展性(增量开发),且使其处理情况更具有一般性(如可处理诸如 $*x+3$ 含指针运算的表达式);本文还最终实现了含指针单子切片器原型.

8 总 结

指针的引入给程序分析带来了3个主要问题. 对于如何表示递归数据结构问题,本文采用了一种简单而又具一定精确度的方法,将所有在同一程序点处申请的堆空间看成一个数组,并将这个数组当成整体处理. 对于如何在数据依赖中包含别名的问题,我们通过指向分析、扩展赋值语句的语法及重新定义引用集来实现泛化数据依赖的. 此外,我们还考虑了赋值语句中左值表达式对变量的可能引用情况. 对于指向分析问题,本文采用了数据流迭代方法来分析指向. 这种方法是流敏感的,具有一定的精度,而且因指向分析和切片计算同时进行,故不需要像一般的流敏感分析方法那样记录每一个程序点的指向信息,而只需记录当前所分析的程序点处指向信息,从而节省了存储空间.

本文将指向分析与已有的单子静态切片方法结合,实现了模块单子切片方法对指针的扩展. 通过将指向分析融入切片计算,我们仍然可以直接在抽象语法项上计算切片. 扩展后的单子切片算法可以很好地融合进单子切片方法中,且扩展后的单子切片方法还继承了原有算法较强的组合性和程序语言适应性.

由于单子语义的可执行性保证了我们切片方法

的可行性.从而单子切片技术能在一定程度上解决目前在动态切片、面向对象程序切片及并发程序切片等方面存在的问题.这也是我们下一步研究工作重点，并打算针对某个具体实际语言(如 Java)进行单子切片研究及其切片工具的开发.

参 考 文 献

[1] Weiser M. Program slicing. IEEE Transactions on Software Engineering, 1984, 16(5): 498-509

[2] Tip F. A survey of program slicing techniques. Journal of Programming Languages, 1995, 3(3): 121-189

[3] Binkley D, Gallagher K B. Program slicing. Advances in Computers, 1996, 43: 1-50

[4] Li Bi-Xing. Program Slicing Technique and Its Applications. Beijing: Science Press, 2006(in Chinese)
(李必信. 程序切片技术及其应用. 北京: 科学出版社, 2006)

[5] Xu Bao-Wen, Qian Ju, Zhang Xiao-Fang, Wu Zhong-Qiang, Chen Lin. A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes, 2005, 30(2): 1-36

[6] Chen Zhen-Qiang. Technical research on program slicing based on dependence analysis [Ph. D. dissertation]. Southeast University, Nanjing, 2002(in Chinese)
(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]. 东南大学, 南京, 2002)

[7] Zhang Ying-Zhou, Xu Bao-Wen, Shi Liang, Li Bi-Xin, Yang Hong-ji. Modular monadic program slicing//Proceedings of the 28th Annual International Computer Software and Applications Conference, COMPSAC 2004. Hong Kong, China, 2004: 66-71

[8] Zhang Ying-Zhou, Xu Bao-Wen. An approach to dynamic program slicing based on modular monadic semantics. Chinese Journal of Computers, 2006, 29(4): 526-534(in Chinese)
(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)

[9] Zhang Ying-Zhou, Xu Bao-Wen. A novel formal approach to program slicing. Science in China, Series F: Information Sciences, 2007, 50(5): 657-670

[10] Zhang Ying-Zhou. Research on program slicing techniques based on modular monadic semantics [Ph. D. dissertation]. Southeast University, Nanjing, 2006(in Chinese)
(张迎周. 基于模块单子语义的程序切片技术研究[博士学位论文]. 东南大学, 南京, 2006)

[11] Liang S, Hudak P, Jones M. Monad transformers and modular interpreters//Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL'95), 1995: 333-343

[12] Liang S. Modular monadic semantics and compilation [Ph. D. dissertation]. University of Yale, Yale, 1998

[13] Wansbrough K. A modular monadic action semantics[M. S. dissertation]. University of Auckland, Auckland, 1997

[14] Hind M, Burke M, Carini P et al. Interprocedural pointer alias analysis. ACM Transactions on Programming Languages and Systems, 1999, 21(4): 848-894

[15] Landi W, Ryder B G. A safe approximate algorithm for interprocedural aliasing//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. San Francisco, CA, 1992: 235-248

[16] Emami M, Ghiya R, Hendren L. Context-sensitive interprocedural analysis in the presence of function pointers//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. Washington, DC, 1994: 242-256

[17] Song Y, Huynh D. Forward dynamic object-oriented program slicing//Proceedings of the Application-Specific Systems and Software Engineering and Technology (ASSET'99), 1999: 230-237

[18] Wadler P. Comprehending monads//Proceedings of the ACM Conference on Lisp and Functional Programming. Nice, France, 1990: 61-78

[19] Zhang Y Z, Labra J, del Río A. A monadic program slicer. ACM SIGPLAN Notices, 2006, 41(5): 30-38

[20] Haskell Website 2009. http://www.haskell.org/, 2009

[21] Zhang Ying-Zhou, Zhang Wei-Feng. Haskell: A modern purely functional programming language. Journal of Nanjing University of Posts and Telecommunications (Natural Science), 2007, 27(4): 13-18(in Chinese)
(张迎周, 张卫丰. Haskell: 一种现代纯函数式语言. 南京邮电大学学报, 2007, 27(4): 13-18)

[22] Jones N D, Muchnick S S. Flow analysis and optimization of LISP-like structures//Muchnick S S, Jones N D eds. Program Flow Analysis, Chapter 4, Prentice-Hall, 1981: 102-131

[23] Horwitz S, Pfeiffer P, Reps T. Dependence analysis for pointer variables//Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation. Portland, Oregon, 1989: 28-40

[24] Chase D R, Wegman M, Zadek F. Analysis of pointers and structures//Proceedings of the SIGPLAN'90 Conference on Program Language Design and Implementation. White Plains, NY, 1990: 296-310

[25] Choi J D, Burke M, Carini P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects//Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages. Charleston, South Carolina, 1993: 232-245

[26] Ghiya R, Hendren L J. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C//Proceedings of the 23th Annual ACM Symposium on Principles of Programming Language. St. Petersburg, Florida, 1996: 1-15

[27] Horwitz S, Prins J, Reps T. Integrating non-interfering versions of programs. ACM Transactions on Programming Language System, 1989, 11(3): 345-387

[28] Burke M, Carini P, Choi J D, Hind M. Flow-insensitive interprocedural alias analysis in the presence of pointers//Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing. Ithaca, NY, 1994: 235-250

[29] Steensgard B. Points-to analysis in almost linear time//Proceedings of the 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages. St. Petersburg, Florida, 1996: 32-41

[30] Shapiro M, Horwitz S. Fast and accurate flow-insensitive points-to analysis//Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France, 1997: 1-14

[31] Huang Bo, Zang Bin-Yu, Wei Jun-Yin, Zhu Chuan-Qi. Context-sensitive interprocedural pointer analysis. Chinese Journal of Computers, 2000, 23(5): 477-485(in Chinese)
(黄波,藏斌宇,韦俊银,朱传琪. 上下文敏感的过程间指针分析. 计算机学报, 2000, 23(5): 477-485)

[32] Lyle J R, Binkley D. Program slicing in the presence of pointers//Proceedings of the 3rd Annual Software Engineering Research Forum. Orlando, 1993: 255-260

[33] Morrison D R. PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM, 1968, 15(4): 514-534

[34] Hind M, Pioli A. Which pointer analysis should I use?//Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). Portland, Oregon, 2000: 113-123

[35] Hind M. Pointer analysis: Haven't we solved this problem yet?//Proceedings of the Program Analysis for Software Tools and Engineering(PASTE'01). Snowbird, Utah, 2001: 54-61

[36] Ramalingam G. The undecidability of aliasing. ACM Transactions on Programming Languages and Systems, 1994, 16(5): 1467-1471

[37] Hind M, Pioli A. Evaluating the effectiveness of pointer alias analyses. Science of Computer Programming, 2001, 39(1): 31-55

[38] Ruf E. Context-insensitive alias analysis reconsidered//Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation. 1995: 13-22

[39] Agrawal H, DeMillo R A, Spafford E H. Dynamic slicing in the presence of unconstrained pointers//Proceedings of the ACM 4th Symposium on Testing, Analysis and Verification (TAV). British Columbia, Canada, 1991: 60-73

[40] Jiang J, Zhou X, Robson D J. Program slicing for C: The problems in implementation//Proceedings of the Conference on Software Maintenance. Sorrento, Italy, 1991: 182-190

[41] Ernst M D. Slicing pointers and procedures. Microsoft Research: Technical Report MSR-TR-95-23, 1995

[42] Lyle J R, Wallace D R, Graham J R et al. A CASE tool to evaluate functional diversity in high integrity software. National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD: Technical Report IR-5691, 1995

[43] Ross J, Sagiv M. Building a bridge between pointer aliases and program dependences. Nordic Journal of Computing, 1998, (8): 361-386

[44] Ouarbya L, Danicic S, Daoudi M et al. A denotational interprocedural program slicer//Proceedings of the 9th IEEE Working Conference on Reverse Engineering. 2002: 181-189

[45] Hausler P A. Denotational program slicing//Proceedings of the 22th Annual Hawaii International Conference on System Sciences, 1989: 486-495

[46] Wu Z Q, Zhang Y Z, Xu B W. Modular monadic slicing in the presence of pointers//Proceedings of the 6th International Conference on Computational Science (ICCS 2006). University of Reading, UK, 2006: 748-756

ZHANG Ying-Zhou, born in 1978, Ph.D., associate professor. His research interests include software formal methods, program slicing.

WU Zhong-Qiang, born in 1983, M.S. candidate. His research interests include program analysis and software formal methods.

QIAN Ju, born in 1981, Ph.D.. His research interests include program analysis and software testing.

ZHANG Wei-Feng, born in 1975, Ph.D.. His research interests include software testing and information retrieval.

XU Bao-Wen, born in 1961, Ph.D., professor. His research interests include programming languages, software formal technologies, and program analysis.

Background

This work is supported in part by the Young Scientist's Foundation of NSFC. It aims to provide some useful formal approaches for program analysis. This paper will contribute to the theory and applications of monadic program slicing for a program with pointers. The research team has published many papers in journals and international conferences.