北京大学

本科生毕业论文

题目：流敏感及上下文敏感的指针分析算法

姓　　名：杨至轩

学　　号：1300012785

院　　系：信息科学技术学院

专　　业：计算机科学与技术专业

研究方向：程序分析

导　　师：熊英飞

二〇一七年五月

北京大学本科毕业论文导师评阅表

学生姓名	杨至轩	学生学号	1300012785	论文成绩	
学院（系）	信息科学技术学院			学生所在专业	计算机系
导师姓名	熊英飞	导师单位/ 所在研究所	软件研究所	导师职称	研究员
论文题目 （中、英文）	流敏感及上下文敏感的指针分析算法 A Flow-sensitive and Context-sensitive Points-to Analysis Algorithm				

导师评语

指针分析是程序分析中最基础，最重要的问题。指针分析经历了几十年的长期发展，几乎所有程序分析算法都最先在指针分析上进行实验，可以说，在指针分析上做出进一步突破非常困难。杨至轩的本科毕业论文挑战了这个困难的问题，将最新的两个指针分析算法的优势进行了结合，算法设计合理。实验表明，精确度至少比其中一个算法要高（另外一个因为缺乏公开实现而没有对比），效率可以接受。总的来说，本文是一篇优秀的本科毕业论文。

导师签名：

年　　月　　日

版权声明

摘要

指针分析是编译优化、程序分析领域重要的基础问题之一。因指针分析问题在可计算性上是不可判定的，所以对该问题的研究的主要角度是算法的“精确度”与算法的“运行效率”之间的平衡。

近四十年来，学界已有大量工作设计不同准确率、不同计算效率的指针分析算法，近年来较有影响力的包括 Lattner 等人的上下文敏感分析算法 DSA[22]，及 Hardekopf 等人的流敏感稀疏分析算法 SFS[17]。本文设计了一种结合两者优点的，首先在函数调用强连通分量内进行基于静态单赋值形式的稀疏流敏感指针分析，然后在函数调用强连通分量之间进行自底向上基于“归纳”的上下文敏感指针分析算法。

本文算法的时间复杂度为 $O(VE\alpha(V))$，其中 V 是语句的个数，E 是程序控制流图中边的条数，$\alpha(\cdot)$ 是阿克曼函数 $A(n,n)$ 的反函数，也在 DSA 算法复杂度 $O(V\alpha(V))$ 与 SFS 算法复杂度 $O\left(V^2E\right)$ 之间。在 SPEC2000 程序集上的实验评估表明，我们的算法能比之前算法有更高的指针分析准确率。

关键词：静态分析, 指针分析, 上下文敏感，流敏感

A Flow-sensitive and Context-sensitive Points-to Analysis Algorithm

Zhixuan Yang (Department of Computer Science and Technology)
Directed by Prof. Yingfei Xiong

ABSTRACT

Points-to analysis is a fundamental problem in the field of program optimizing and static program analysis. Since precise static points-to analysis is undecidable theoretically, research on this problem focuses on the balance of efficiency and precision of practical algorithms.

In the past 40 years, a lot of algorithms with different precision and time-complexity were proposed. Recently, Lattner et.al.'s context-sensitive algorithm DSA[22] and Hardekopf et.al.'s flow-sensitive algorithm SFS[17] are influencial in this field. This dissertation proposes a new algorithm combining the advantages of both DSA and SFS. Our algorithm performs sparse data-flow analysis on the static single assignment form in the level of strong-connected components of the call graph, and then performs bottom-up summary inlining between strong-connected components of the call graph.

The time-complexity of our algorithm is $O(VE\alpha(V))$, where V is the number of instructions, E is the number of edges of control-flow graph and $\alpha(\cdot)$ is the inverse of the Ackermann funtion $A(n, n)$. The complexity of our algorithm lies between the complexity of DSA $O(V\alpha(V))$ and the complexity of SFS $O\left(V^2E\right)$. Experiments on the SPEC2000 program set shows our algorithm out-performs DSA in precision of points-to analysis.

KEYWORDS: Static Analysis, Points-to Analysis, Context-sensitivity, Flow-sensitivity

目录

第一章　序言

指针是一类用于对程序中变量进行指代、引用的编程语言构造，其在不同范式的众多编程语言中都广泛地存在。在 C 语言中，指针是语言的“一等公民”，程序员可以自由创建指针、使指针指向不同的变量、并通过指针访问其所指向的变量。在 Python, Javascript 等语言中，指针①则被隐式地广泛使用，程序员创建的每个变量概念上都是指向某个对象的指针，且程序员只能通过这些隐式的指针访问变量。在一些函数式编程语言，如 ML 语言中，也提供指针的构造②，不过语言中“纯”（函数式）的变量与“不纯”的变量被区分开，指针只能指向不纯的变量。甚至在 Haskell 这样的纯函数式编程语言中也有 Lens[9] 这样可以当做指针的对应物的语言结构。

指针的广泛存在给编译优化、静态程序分析带来困难和挑战。比如在下面这个 C 语言代码片段中，

```
for(i = 0; i < *p; i++) {
    a[i] = 0;
}
```

编译器不能简单地将代码变换为：

```
int n = *p;
for(i = 0; i < n; i++) {
    a[i] = 0;
}
```

的形式。因为如果 `a+i` 和 `p` 指向同一片内存的话，这两段程序的效果是不一样的，于是编译器需要先对程序中指针变量能指向的位置进行分析（即“指针分析”）后才能再进行很多有效的代码优化。指针分析在众多静态软件分析问题中也扮演着基础性的角色，如对 C 语言程序进行静态的内存泄漏检查的问题中，也需要知道程序中指针变量可能指向哪些堆上的内存对象，以及哪些堆上的内存对象已经通过指针被释放。

从 1980 年 Weihl 等人第一次提出指针分析问题[35] 至今，学界对指针分析问题进行了大量研究。其中近年来较有影响力的包括 Lattner 等人的 Data Structure Analysis 分析算法（DSA）[22]。DSA 是一个上下文敏感、域敏感（field-sensitive）的分析算法，其与更早研究成果的主要区别是该算法对堆上的内存对象的建模方式不是简单将每一条分配语句（如 C 中的 `malloc()` 语句）抽象为一个堆对象，而是把在不同函数调用上

① 在此类语言中，往往称之为“引用”而不是“指针”。本文则一律称作“指针”。
② 即引用类型 `ref T`。

下文下被执行的同一条分配语句当做不同的内存对象（称之为 heap cloning，或 heap specialization[24]，本文中称作“堆上下文敏感”），也就是说 DSA 能够区分在不同位置通过同一辅助函数被创建的内存对象。DSA 是第一个在无环调用路径上能实现完整 heap cloning 的高效分析算法。

近年来另一有影响力的研究成果是 Hardekopf 与 Lin 发展的基于子集的流敏感指针分析[14][15][18][16][17]。其最先进的成果 Staged Flow-sensitive Analysis (SFS)[17] 是一个流敏感、基于子集、但上下文不敏感的高效分析算法。

Lattner 等人的 DSA 分析算法是是流不敏感、上下文敏感、堆上下文敏感的基于合一（unification-based ）指针分析，而 Hardekopf 等人的 SFS 则是一个流敏感，但上下文不敏感、堆上下文不敏感的基于子集（subset-based）指针分析。可以看出，这两个算法对“指针分析”这同一问题的处理颇为不同，但得到的算法的优势恰是互补的。那么一个自然的问题是，能否设计出一个兼具两者优点的高效指针分析算法呢？从这个角度出发，本文作者在单调数据流框架下对这两个算法进行了分析，并设计出一个兼具两者优点的，即上下文敏感、流敏感、且堆上下文敏感的指针分析算法。

我们的算法在过程内使用类似于 SFS 的方式进行流敏感的指针分析，并得到整个函数对内存的操作的一个“归纳”（Summary，相当于 DSA 对函数进行局部分析的结果），然后我们再按照函数调用图将各函数的归纳内联到它的调用者中，并对调用者的数据流分析结果进行更新。因为采用内联支持上下文敏感的方法要求局部分析的单调性，而流敏感分析中对 strong-update 的特性违反了单调性，故我们对函数参数、全局变量放弃 strong-update，对其余变量仍支持 strong-update。另外，我们在函数调用图的强连通分量内放弃上下文敏感性以使我们避免进行递归地内联。本算法与 DSA 算法在实验评估中的结果表明，我们的流敏感分析算法能有更高的指针分析的准确率。

本文余下部分按如下方式组织：

- 第二章定义指针分析研究中几个“敏感性”的概念以及与本文相关的研究工作。
- 第三章介绍我们设计的流敏感、上下文敏感、堆上下文敏感分析算法。
- 第四章对本文提出的算法进行实验评估。
- 第五章总结全文并预测未来的研究方向。

第二章　指针分析中的“敏感性”

指针分析的目标是对程序中指针变量可能的取值进行静态分析。由计算复杂性中的 Rice 定理[32] 可知，精确的指针分析是可不判定问题。所以现实的指针分析算法皆需要对精确结果进行某种近似，以得到能有效中止的算法。一般来说，指针分析算法进行的是“上近似”，也就是算法输出的结果总是包含真实的精确结果。

各类指针分析算法所采取的近似方法一般可以按如下几个维度进行分类：

- 赋值方向性，即是基于合一还是基于子集。
- 上下文敏感性。
- 对堆的建模方式。
- 流敏感性。
- 域敏感性。

2.1　赋值方向性

基于合一的分析方法的代表性工作是 Steensgaard 算法[33]。基于合一指的是，如果两个变量可能被同一个指针指向，那么这两个被指向的变量被视作“等价”，分析认为任何可能指向其中某一个变量的指针也可能会指向另一个。所以对基于合一的分析算法来说，一个指针最多指向一个变量的等价类。显然这样的近似策略会得到比精确结果更大的答案，然而这样的策略使得分析算法可以使用 Tarjan 等人的 Union-Find[10] 数据结构维护变量的等价类，使得任何类型的语句都可以在常数时间处理。Steensgaard 算法同时也是上下文不敏感且流不敏感的，所以拥有对被分析语句条数线性时间的复杂度。Steensgaard 算法是第一个可以进行大规模分析的指针分析算法，在实践中得到大量使用。

基于子集的分析方法的代表性工作是知名的 Andersen 算法[1]。基于子集即是指没有采用基于合一的近似方法，也就是每个指针可以指向任意数量的变量。于是对于程序中形如 `p = q` 的语句，基于子集的算法认为这条语句的效果是 `q` 可能指向的变量的集合是 `p` 所可能指向的变量的集合的子集。而基于合一的算法认为此条语句的效果是 `q` 可能指向的变量的集合是与 `p` 所可能指向的变量的集合是相等的。因此指针分析是基于子集还是基于合一的属性也被叫做“赋值方向性”。

2.2 上下文敏感性

上下文敏感性是静态程序分析中广泛存在的概念，其指的是在进行过程间程序分析（inter-procedural program analysis ）时，分析算法考虑的被分析程序中的控制流路径是否都是满足正确的调用－返回匹配关系的。比如在图2.1中，有两处不同的对函数 `f` 的调用。上下文敏感分析算法所考虑的程序执行路径只包含那些正确的调用－返回路径，比如从 A 点调用 f 则从 f 返回时总是回到 A 点。而与此相对的上下文不敏感分析算法则允许不正确的调用－返回路径（unrealizable path ），比如 A 点与 B 点都调用 f，分析算法却认为存在一条路径是从 A 到 f 再返回 B，比如图2.1被标红的一条路径。

上下文不敏感数据流分析算法拥有更简单的实现方式，只需将所有函数的控制流图合并为一个图，然后在函数调用语句与被调用函数入口之间添加边，以及函数返回语句与本函数所有的调用者之间添加边，得到一个超级控制流图（super CFG ），然后在 super CFG 上进行普通的数据流分析即可。

上下文敏感分析算法则需要更复杂的实现方式。最简单的实现方法是“函数克隆”的方法，即对于所有的函数调用语句，都把被调用函数的控制流图复制一份替代这条函数调用语句。但因为被调用的函数也可能会调用其他函数，甚至还会有递归函数调用，所以克隆被调用者时，对于被调用者内的调用语句，不能无限制地进行递归的克隆，而必须限制克隆的深度，克隆的深度达到一定的界限时，则退回到上下文不敏感的方法，所以函数克隆的方法不是完全上下文敏感的，只是部分上下文敏感的。

上下文敏感分析还有更有效的实现方式。包括**函数归纳方法**（summary-based method ）[31] 以及 **CFL 可达性方法**（context-free-language reachability method ）[28] 这两种通用的实现方法。函数归纳方法的直观思路是对于每个函数计算一个“归纳”，能代表整个执行整个函数的效果，而对于调用这个函数的调用语句的效果则是“应用”这个归纳。因为递归调用的存在，“归纳”的计算可能需要是迭代计算的。而 CFL 可达性方法的做法则是首先将程序分析问题规约为另一个图中的可达性问题（从一个顶点能否通过边到达另一个顶点），而正确的调用－返回匹配关系的要求则被归纳为在这个图中的边上添加了一些标记，并且增加要求为图中的 CFL 可达性问题，即是问“从一个顶点能否通过边到达另一个顶点，且路径上的标记形成的字符串属于某个上下文无关文法”。

CFL 可达性方法适用于满足单调、可分配性质的数据流分析问题，但指针分析往往不完全具有这样的性质，故在作者所知范围内，还没有看到用 CFL 可达性方法实现上下文敏感指针分析的方法。函数归纳方法则可以被扩展到更广泛的情况，因此在上下文敏感指针分析中得到广泛的应用。[36][20][22][37]。

DSA 也通过函数归纳法支持上下文敏感性。DSA 出于算法效率考虑，对于被分析程序函数调用图中处于同一个强连通分量的函数（直接或间接地递归调用的函数）放

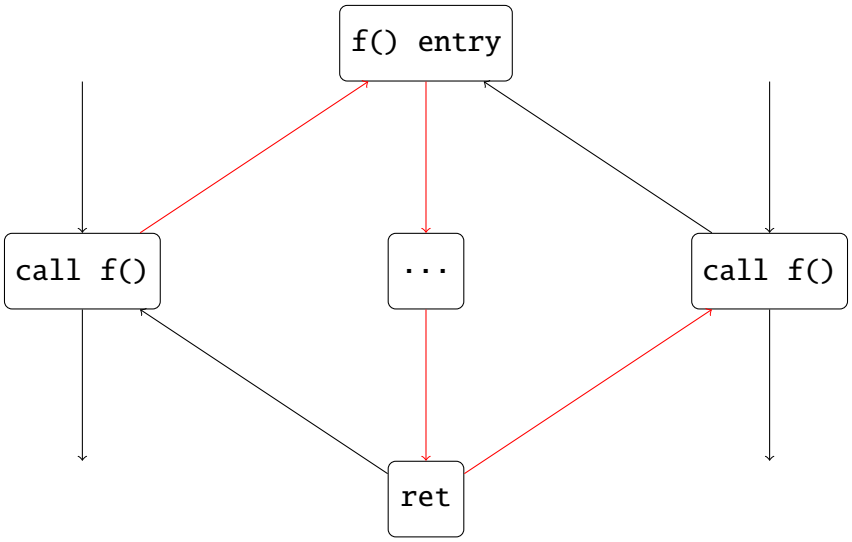


图 2.1　错误的调用－返回匹配路径。上下文不敏感算法允许这样的路径存在。

弃他们之间的上下文敏感性。所以 DSA 的算法思路是先在函数强连通分量内进行局部的流不敏感的基于合一的指针分析（类似于 Steensgaard 算法），把结果当做此函数强连通分量的归纳。然后对函数强连通分量的无环调用图按照后序遍历的次序将被调用者（callee ）的归纳应用到调用者（caller ）。

2.3　堆的建模方式

指针分析精确性的另一个维度是算法如何对堆上的对象进行处理。最简单且被广泛使用的做法是把一条堆分配语句当作一个堆资源，而此条语句被多次执行都被当作返回的是同一个资源，这样的近似方法被称作**一次性分配抽象**（one-time allocation abstraction）或 **按分配位置抽象方法**（allocation site abstraction method)。一次性抽象方法对指针分析准确率可能造成较大的影响，按照实践经验，现实程序可能会尽量避免直接使用堆分配函数，而使用被包装过分配版本。比如来自 SPEC2000 程序集中的 `gcc` 编译器源代码定义了图2.2中的辅助函数，然后整个项目的其他位置没有再对 `malloc` 的调用，而全部调用 `xmalloc` 。如果使用一次性抽象，则指针分析会认为整个程序中只有一个堆上的对象，对分析精度造成了很大的影响。

Nystrom 等人[24] 首先指出了需要对同一条分配语句在不同情况下的执行的返回结果进行区分的重要性，他们验证了如果能把分配语句执行时的函数调用上下文进行区分（Nystrom 等人称为 heap-specialization ），那么可以显著改善指针分析结果。DSA 则是第一个实现高效的“无限制”深度 heap-specialization 的分析算法。所谓“无限制”深度指的在放弃函数调用强连通分量内的上下文敏感性的条件下，一个分配语句被执行时的整个调用上下文都会被区分。

```
/* Same as ‘malloc’ but report error if no memory available.  */

char *
xmalloc (size)
unsigned size;
{
        register char *value = (char *) malloc ((size\_t)size);
        if (value == 0)
        fatal ("virtual memory exhausted");
        return value;
}
```

图 2.2　gcc 中的内存分配辅助函数

2.4　流敏感性

流敏感性指的是分析算法是否对被不同程序点给出不一样的分析结果。流不敏感分析算法只给出一个全局的分析结果，该结果对于任意程序点都是安全（sound ）的。流敏感分析算法则对不同的程序点给出不同的分析结果。流敏感分析的标准做法及理论框架是单调数据流分析框架，在此框架内分析过程相当于在忽略分支条件下不断模拟执行程序。而只要待分析内容是一个有限半格的元素，那么这样的模拟执行总是收敛。流不敏感分析算法的通用做法则是遍历待分析程序中的所有语句（可以按照任何次序），对每条语句按照其语义生成一条关于待分析内容的约束，然后根据所有的约束求解出对于任何一个程序点都安全的分析结果。

显然流敏感分析相比于流不敏感分析能得到精确得多的结果，但其所需的计算量也远远高于流不敏感分析。所以早期流敏感的指针分析[20] 没有在实践中得到大规模的使用。Hardekopf 等人在 2009 年[18] 及 2011 年[17] 的研究工作首次实现了高效的流敏感分析算法，其性能达到了可分析百万行代码的级别。他们的研究中提高流敏感分析性能的关键方法是进行“稀疏分析”[5]，即首先把程序转换为**完全静态单赋值形式**，然后使得每个程序点只储存记录本条语句可能修改的指针的指向集合，并且数据流按照静态单赋值形式中指针的“定义-使用关系”（def-use relation）进行传播。

Hardekopf 等人在 2011 年[17] 的较新方法是“分阶段流敏感指针分析”（Staged Flow-sensitive Analysis, SFS）。此算法的流程是首先进行任意一个流不敏感的指针分析，根据此分析的结果可以知道每条语句可能修改哪些指针的取值，根据这样的信息我们可以将代码转换为静态单赋值形式[8]，有了静态单赋值形式的程序后，我们可以知道每条语句使用的变量的上一次修改在什么位置，所以可以每条语句只保存它可能会修改的指针的指向集合，然后直接沿着“定义－使用边”进行数据流传播。

2.5　域敏感性

域敏感性是关于指针分析如何处理“结构体（struct）/ 记录（record）/ 类（class）”这样用户自定义聚合类型的性质。域不敏感指的是分析算法不区分结构体的各个域（field），而只分析一个结构体对象从它的任意一个域可能指向哪些对象。而域敏感分析则可以区分结构体中不同的域分别可能指向哪些对象。

显然对于 C++，Java 等大量使用聚合数据类型的面向对象编程语言域敏感性尤其重要。但 Reps[29] 证明了在进行过程间分析时，同时是完全的上下文敏感与完全的域敏感分析问题是不可判定的。于是对指针分析算法的一个重要问题是如何在上下文敏感性与域敏感性平衡。研究[26] 的实验表明对于 C 语言等面向过程的编程语言中，上下文敏感性对分析精度的影响较大，而对于 Java 等面向对象编程语言中，域敏感性对分析精度的影响较大。

2.6　相关工作总结

图2.3分类汇总了各类精确度的指针分析算法。可以看出，对指针分析的研究已经从上世纪 90 年代的追求实践中可用的高效算法转变为近年来在高效的前提下提升精度。而我们的算法则是已知工作中精度最高的。

图 2.3　指针分析算法相关工作汇总

	基于合一	基于子集	流敏感
上下文不敏感	• Weihl 1980[35] 第一篇指针分析的研究工作 • Steensgaard 1996[33] 第一个高效的分析算法	• Andersen 1994[1] • Hardekopf 2007[14] 进行动态 SCC 检测优化	• Choi 1993[4] • Hardekopf 2009[18] 2011[16] 基于子集
上下文敏感	• Lattner 2007[22] 且是域敏感、堆上下文敏感的	• Whaley 2004[36] • Nystrom 2004[23] 且是堆上下文敏感的 • Sui 2014[34]	• Zhu 2005[39] • Kahlon 2008[19] • **本文的算法** 且是堆上下文敏感的

第三章　上下文、流、堆敏感指针分析算法 CFHS

本章介绍我们设计的上下文敏感、流敏感、堆上下文敏感基于合一的指针分析算法，简便起见，我们称之为 CFHS 算法（Context-Flow-Heap-Sensitive Algorithm）。CFHS 算法是一个分阶段的算法，其大致流程为：

1. 其首先使用其他任意指针分析（称之为“辅助分析”）对程序进行快速而粗略的指针分析。

2. 根据辅助分析的结果将程序转换为完全静态单赋值形式，得到“定义－使用图”（Def-Use Graph, DUG）。

3. 根据辅助分析的结果，识别出函数调用图。并且对函数调用图中的每一个强连通分量（Strong Connected Component, SCC ），将每个函数的 DUG 拼接合并为 SCC 级别的超级 DUG。

4. 对于每个 SCC 的 DUG，进行稀疏流敏感指针分析，并对每个函数生成“归纳”。

5. 在 SCC 的无环调用图中，后序地处理每个 SCC，对每个调用指令，将被调用函数的归纳应用到调用指令处，并更新 SCC 的流敏感分析结果及摘要。

在本章接下来的各节中，我们详细说明算法的每一个步骤，并给出伪代码及实例说明。

3.1　内存 SSA 变换

所谓静态单赋值形式（SSA）[8] 指的是程序中每一个变量只被定义恰好一次。变量可以被多次定义的原始形式的程序被转换为 SSA 形式后，被多次定义的原始变量被分割为不同的实例，每个实例对应一次定义。在程序控制流图的交汇点，同一个变量的不同实例通过 ϕ 函数进行合并，产生一个这个变量的一个新的实例。SSA 形式非常适于进行稀疏数据流分析，因为它显式地表示出了“定义－使用”关系，使得数据流信息可以直接沿着变量定义－变量使用边进行传播[27]。图3.1是一个原始形式程序与相应的 SSA 形式的例子。

```
a = 0;
if(...) {
    a = 1;
} else {
    a = 2;
}
```

(a) 原始形式

```
a0 = 0;
if(...) {
  a1 = 1;
} else {
  a2 = 1;
}
a3 = φ(a1, a2);
```

(b) SSA 形式

图 3.1　SSA 形式示例

3.1.1　辅助指针分析

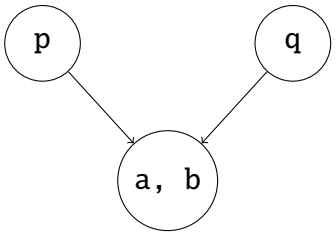
指针的存在使得SSA转换变得困难，所以现代编译器的中间表示格式，比如LLVM IR[21]，往往是“部分 SSA 形式”，即被取过地址的变量仍然是可修改的内存中的变量形式，而没有被取过地址的变量则可以被优化为静态单赋值形式。而如果我们想把程序中的所有变量转换为SSA形式，则必须要进行指针分析，从而知道通过指针进行的间接访问修改的是哪些变量。但因为指针分析生成的是“保守”的结果（即指针分析结果认为p指向a，实际上p不一定真的会指向a，但指针分析一定包含真实的结果），所以通过指针分析识别出的间接访问、修改实际只是“潜在“间接修改。我们也采用Chow等人[6]的方法，对于间接写入（如`*x = y`）我们会把其视作对`x`所有可能指向的`a`进行既读取又修改，对于间接读取（如`y = *x`），我们把其当作当作对`x`所有可能指向的`a`进行了读取。

辅助分析的选择只要是安全的即可，Hardekopf等人的研究工作[16]中选择的是流不敏感、上下文不敏感的基于子集的分析算法（Andersen算法）。而在我们的研究工作中，我们选择使用Lattner等人的DSA算法。DSA算法相对于Andersen算法效率更高，且因为DSA算法是基于合一的，所以在分析结果中，每个指针总是指向最多一个“变量等价类”，所以我们在构造SSA形式时，可以以变量等价类为基本单位，从而简化算法并在某些情况下改进算法的效率。而如果使用Andersen算法，则需要像SFS算法一样进行额外的识别变量等价类的步骤（16中的Access Equivalence优化）。

我们通过图3.2中的程序对我们的转换方法进行示例。在图3.2a的程序中，指针`p`既可能指向`a`也可能指向`b`，所以对于流不敏感、基于合一的DSA会将`a`和`b`视为一个“变量等价类”，并且`q`也指向这一个等价类。根据辅助指针分析提供的信息，我们可以知道原始程序每条语句都直接或间接地修改哪些变量。图3.3显式地在程序中标注出了通过指针进行间接访问的变量。按照Chow等人的记号[6]，我们对间接读取用 $\mu()$ 、间接写入用 $\chi()$ 这两个虚拟函数表示。

```
int a, b;
int *p, *q;
if(...) {
    p = &b;
    q = &a;
    *p = a;
    b = *q;
} else {
    p = &a;
    *p = 0;
}
a = b;
```

(a)



(b)

图 3.2　左图：一段示例程序。右图：DSA 生成的这段程序的指向图。

```
int a, b;
int *p, *q;
if(...) {
    p = &b;
    q = &a;
    *p = a; ab = χ(ab);
    b = *q; μ(ab);
} else {
    p = &a;
    *p = 0; ab = χ(ab);
}
a = b;
```

图 3.3　显式标注了通过间接读取修改的变量的程序

3.1.2 $\phi()$ 函数放置

在知道每条语句可能会修改哪些变量后，我们可以使用标准算法[2][3][7][8] 将代码转换为静态单赋值形式。我们实现的算法按如下几个步骤进行：

1. 对每一条修改变量的指令放置一个 ϕ 指令在支配边界处。在我们的实现中，我们直接使用 LLVM 提供的支配树分析结果来获取每个基本块的支配边界。算法伪代码见图A.1。注意我们对分配变量的指令 `AllocInst` 理解为它也修改了它分配的变量的值为“任意值”。

2. 然后我们对所有新添加的 ϕ 节点也在它们的支配边界放置新的 ϕ 结点，直到所有的 ϕ 结点都被这样处理。如图A.2。

3. 遍历程序，对每条访问内存变量的指令解析出它访问的变量的上一次“定义”在哪里。方法是：按直接支配关系遍历每个基本块，同时维护每个变量的最后一次定义的位置。对于每个基本块，首先对基本块内每条指令根据维护的“最后一次定义位置”解析出它们的“定义－使用边”，然后对这个基本块在 CFG 中的所有后继基本块，更新后继基本块开头的 ϕ 结点的参数，最后递归处理基本块在支配树上的儿子基本块（既被本基本块直接支配的基本块）。算法如图A.3。

对于图3.3中的程序，经过 SSA 放置算法处理之后的结果可参见图3.4。图中蓝色线是内存变量的定义使用边。为清晰起见，图中只画出了 ab 这一个变量等价类的边。灰色边和黑色边是控制流边。虚线边连接的是每个指针与 DSA 分析结果中它指向的变量等价类之间的边。

3.1.3 过程间内存 SSA

前两小节的算法描述中略过了对函数调用指令的处理方法。因为后续的指针分析阶段是上下文敏感的，所以我们必须在内存 SSA 转换时做好准备。对于函数调用指令，我们在进行内存 SSA 转换时，假设函数调用可能会修改所有通过参数传递的指针能指向或间接指向的内存，所以我们也在函数调用处为这些内存每个创建一个 ϕ 结点，并当作是对这个变量的一次读写。由于函数还可能会通过参数或返回值返回一些新的内存变量，并且调用者在函数调用之后也可能读写这些变量，所以我们也必须对这些新创建的变量创建相应的 ϕ 结点，并当作一次新的定义。

在后续的指针分析中，对于函数调用我们可能有两种处理方法：

- 上下文敏感方式，用于处理没有互相递归调用的函数。

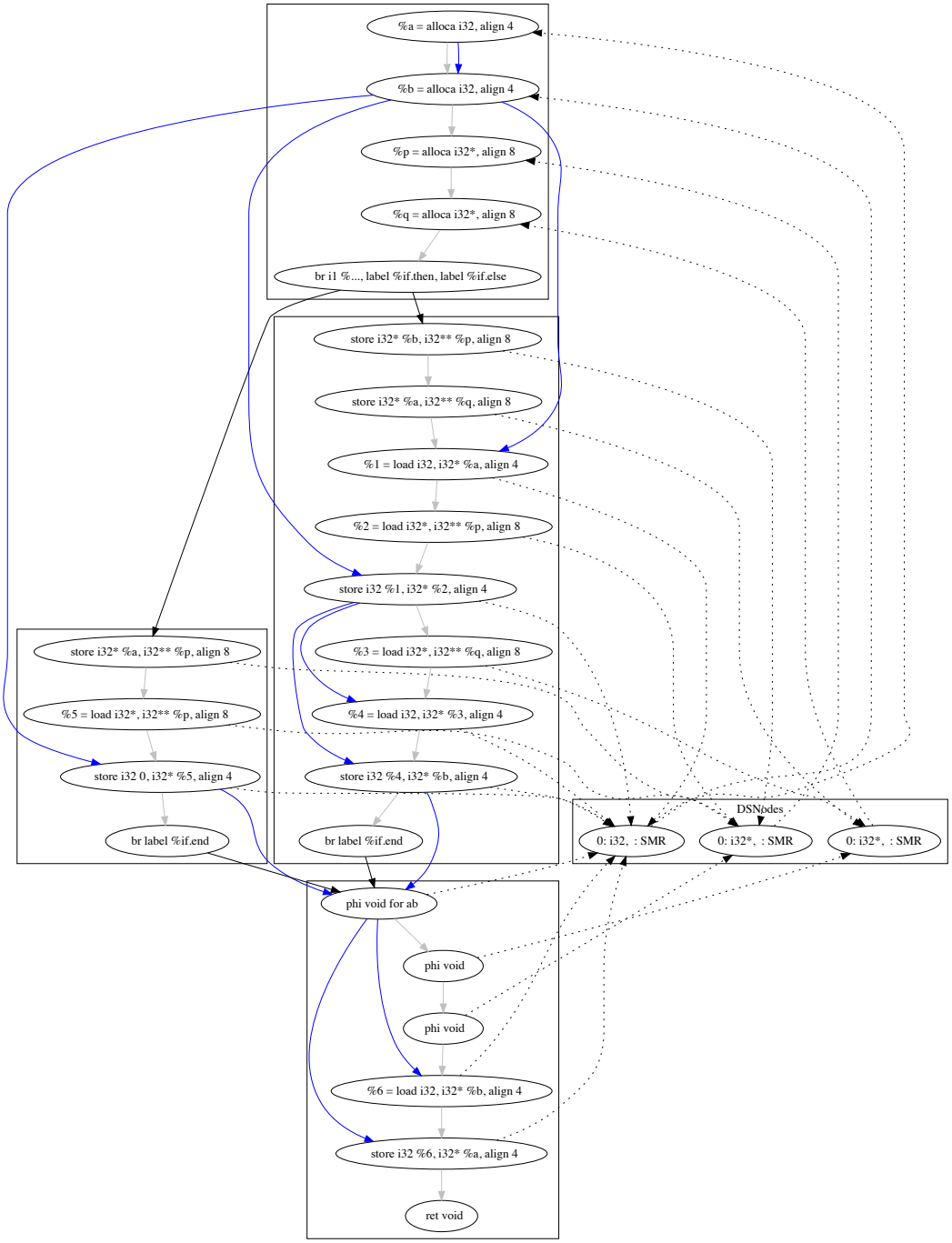


图 3.4　内存 SSA 转换的结果，图中蓝色线是内存变量的定义使用边。为清晰起见，图中只画出了 ab 这一个变量等价类的边。灰色边和黑色边是控制流边。虚线边连接的是每个指针与 DSA 分析结果中它指向的变量等价类之间的边。

- 上下文不敏感方式，用于处理互相递归调用的函数。这要求我们能够将不同函数的 SSA 形式合并拼接在一起，并处理好参数的传递。所以我们还需要在内存 SSA 阶段进行如下准备：对每个能通过函数参数直接或间接指向的变量等价类，在函数开头创建一个 ϕ 结点并当作对这个变量等价类初始的定义。对于每个能通过函数参数或返回的指针直接或间接指向的变量等价类，记录它在函数体内最后一次定义的位置。对于每个函数调用指令，记录通过传递的实际参数能够直接或间接地指向的变量等价类的最后一次定义位置。算法伪代码参考附录中的图A.4。

3.2 局部分析阶段

我们的分析算法是在函数调用强连通分量内部放弃上下文敏感性，然后再在强连通分量之间的无环调用图上做自底向上的内联。所以我们可以先在每个强连通分量间进行局部的分析。所谓局部的分析指的是在这一阶段的分析过程中，对于 SCC 外的函数调用不进行处理。但因为间接函数调用的存在，所以识别出函数调用图和指针分析是互相依赖的，在我们的实现中，我们直接使用 DSA 的分析结果来识别函数调用图中的强连通分量。局部的分析按如下几个步骤进行：

1. 解析强连通分量内的函数调用。因为我们在内存 SSA 转换阶段已经为每个函数调用记录了实际参数的定义位置，函数参数返回的“合并点”，且为每个函数参数在函数开头创建了 ϕ 结点作为“定义点”以及每个参数和返回值记录了最后一次定义位置。对于 SCC 内的函数调用，我们只需要将这些相应的定义点和合并点创建“定义－使用边”连接起来即可。

2. 识别出 SCC 内的所有内存资源，包括函数参数，堆栈上的变量，全局变量。

3. 对于 SCC 内所有和内存与指针相关的指令，创建出它们的“定义使用图”，图中的结点是那些和分析相关的指令，图中的边包括 SSA 转换得出的内存“定义使用边”，以及 LLVM 的部分 SSA 形式的“定义使用边”。

4. 把定义使用图中的所有结点，按照支配顺序初始化工作表（Worklist ）。

5. 在工作表上不断迭代应用每个结点对应的指令。如果在该点的分析结果有变化，则把它的所有后继结点放入工作表。

在上述步骤的第4步中，我们需要按照支配顺序（也就是指如果结点 a 在控制流图中支配 b，那么 a 在工作表中比 b 靠前），而不能像普通的数据流分析中按照任意顺序

进行初始化的原因是对于支持 strong-update 的内存修改指令，它对应的转移函数严格上不是单调的。比如说，对于指令 `store p, a`（把 a 保存到 p 指向的内存中），如果 a 也是指针，p 是二级指针，假设一个指向图 P1 是：p 不指向任何变量，a 指向 z，x 指向 y，那么这条指令的转移函数对于此输入得到的输出应该和输入一样，因为 p 不指向任何东西。而假如指向图 P2 是: p 指向 x，其余和指向图 P1 一样，那么指令的转移函数作用在 P2 上的结果应该是使得 x 指向 z，其余和 P2 一样（因为支持 strong-update，而 p 唯一指向 x，所以 x 被改为指向 z）。可以看出 P1 是 P2 的子集，但 P1 的结果却不是 P2 的子集，故 strong-update 的转移函数不是单调的。但实际上，只要对于所有 p 指向集合不是空集的输入，strong-update 也是单调的，所以我们在初始化时总是按照支配顺序进行初始化，使得所有 StoreInst 在被计算时指针参数总是已经被计算过了。

3.2.1　指向图半格

步骤5中进行的数据流分析在每个结点处保存的是一个“部分指向图”（partial points-to graph），如定义1。在实现中我们可以用并查集来实现变量之间的等价关系。称之为“部分”指向图的原因是在每一个结点的指向图永远只会储存这条指令涉及的指针指向的变量的相关信息。

定义 1 一个指向图（$\mathcal{PG}$）是二元组 (E, P)，其中 E 是各个内存变量的等价关系，P 是变量等价类上的指向关系。且 P 是一个偏函数（即如果 $[x]P[y]$（$[x]$ 指向 $[y]$）且 $[x]P[z]$，那么 $[y] = [z]$）。

数据流分析中的交汇操作是指向图的“合并操作”，如定义2。

定义 2 两个指向图 (E_1, P_1) 与 (E_2, P_2) 的合并的结果是最小的满足如下要求的指向图 (E, P)：

- 若 xE_1y 则 xEy，若 xE_2y 则 xEy。
- 若 $[x]_{E_1}P_1[y]_{E_1}$，则 $[x]_EP[y]_E$。
- 若 $[x]_{E_2}P_2[y]_{E_2}$，则 $[x]_EP[y]_E$。
- (E, P) 是满足定义*1*要求的指向图。

对于我们基于并查集实现的指向图，上述合并操作可以按如下方式计算：先令 $E = E_1$ 及 $P = P_1$。对 E_2 中每一条等价关系 xE_2y，添加到 E 中，然后对于 P_2 中的每一条指向关系 $[x]P_2[y]$，如果不存在 z 使得 $[x]_EP[z]_E$，那么添加 $[x]P[y]$ 到 P 中，如果存在 z 使得 $[x]_EP[z]_E$，则添加 zEy 到 E 中。

按定义1定义的指向图和按定义2定义的合并操作构成了一个半格。这是我们进行流敏感指针分析的基础。

3.2.2 各类指令的转移函数

数据流分析对各种类型的语句按照其语义对应一个指向图上的函数，用于表示这条语句对指向图可能进行的操作。LLVM IR 中只有 StoreInst 和 LoadInst 以及 AllocaInst 是会操作内存的指令，需要我们进行比较复杂的处理，其余的指令要么与指针与内存无关，要么是在进行指针算术，需要我们做的比较简单。接下来我们一一说明对各类指令的处理方法。

对于 LoadInst 指令 `a = load p`，我们只处理 `a` 也是指针，即 `p` 是二级指针的情况，对于情况则是对非指针值的加载，与指针分析无关。对输入指向图 $(E, \rightarrow)$, 如果在其中存在 $[x]$ 使得 $[p] \rightarrow [x]$，且存在 $[v]$ 使得 $[x] \rightarrow [v]$, 那么转移函数的输出是指向图 $(E_v, \{([a], [v])\})$, 其中 E_v 是满足如果 yEv，那么 yE_vv 的最小等价关系，图3.5是这种情况下转移函数的输入输出的示例图。如果不存在 $[x]$ 使得 $[p] \rightarrow [x]$ 或不存在存在 $[v]$ 使得 $[x] \rightarrow [v]$，那么输出是指向图 $(\emptyset, \emptyset)$。ti

图 3.5 LoadInst 的转移函数，左边是输入的指向图，右边是输出的指向图。

对于 StoreInst 指令 `store p, a`, 我们只处理 `a` 也是指针，即 `p` 是指向指针的指针的情况，其余情况则是对非指针值的加载，与指针分析无关。对输入指向图 $(E, \rightarrow)$, 如果不存在 x 使得 $[p] \rightarrow [x]$ 或不存在 x 使得 $[a] \rightarrow [x]$，那么输出和输入一样，为 $(E, \rightarrow)$。如果存在 x 使得 $[p] \rightarrow [x]$ 且存在 v 使得 $[a] \rightarrow [v]$，那么继续分如下两种情况：

- $[x]$ 这个等价类只有一个元素，那么我们进行 strong-update ，输出为 $(E, \Rightarrow)$。其中 $\Rightarrow'$ 为 ‘ $\rightarrow'$ $\{([x], \_)\} \cup \{([x], [v])\}$。意思删除原指向图中 $[x]$ 的指向关系，然后添加指向关系使得 $[x]$ 指向 $[v]$。图3.6是这种情况图形化的说明。

- $[x]$ 这个等价类有多个元素，那么我们进行 weak-update。如果存在 u 使得 $[x] \rightarrow [u]$, 那么输出为 $(E', \rightarrow)$，其中 E' 是在 E 上添加 u 与 v 等价的等价关系。如果不如果存在 u 使得 $[x] \rightarrow [u]$，那么输出为 $(E,' \rightarrow' \cup \{([x], [v])\})$。意思是如果 $[x]$ 已经指

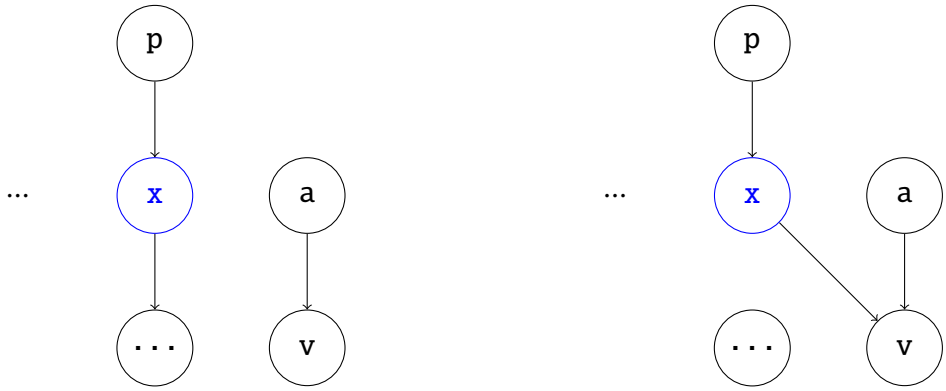


图 3.6　StoreInst 在进行 strong-update 情形下的转移函数示例图，左边是输入的指向图，右边是输出的指向图。其中 x 的变量等价类集合只有唯一的元素 x。

向了一个 $[u]$，那么我们把 $[u]$ 和 $[v]$ 这两个等价类合并，使得 $[x]$ 既指向原来的 $[u]$ 也指向原来的 $[v]$。如果 $[x]$ 还不指向任何东西，那么我们使 $[x]$ 指向 $[v]$ 即可。图3.7是这种情形下的转移函数图示。

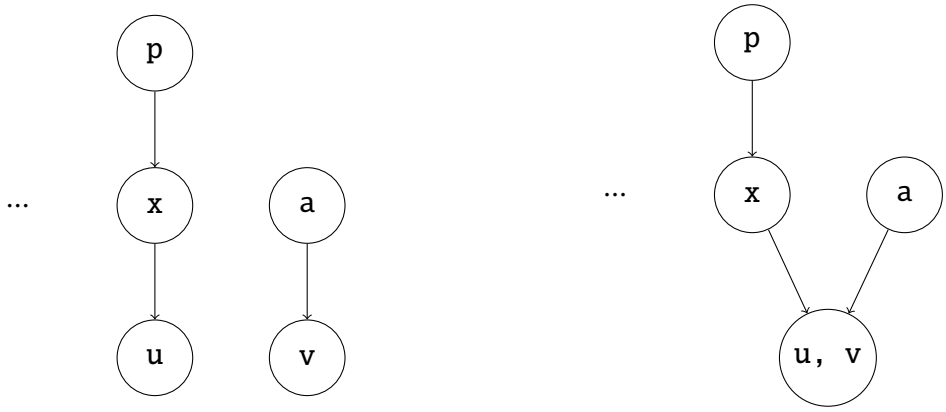


图 3.7　StoreInst 在进行 weak-update 情形下的转移函数示例图，左边是输入的指向图，右边是输出的指向图。

对于 AllocaInst 指令，我们把它视作向它分配的变量写入一个特殊的“任意位置指针”的 StoreInst 指令。处理方法与 StoreInst 相同。其余类型的指令皆不操作内存，但可能进行指针算术，比如：

LLVM 中的类型转换指令 `q = bitcast p`, 因为我们的分析不跟踪类型信息，所以输出只要把输入的指向关系中的 $[p]$ 替换成 $[q]$ 即可。

LLVM 中用于用于计算数组元素地址、结构体域地址的指针算术指令 `q = getelementptr p, ...`，因为我们进行的是域不敏感分析，所以我们将结构体的每个域、数组的每个元素都视作同一个内存对象，因此输出也只要把输入的指向关系中的 $[p]$ 替换成 $[q]$ 即可。

对于 PHINode 指令 `a = phi(a1, a2, ..., ak)`，如果转移函数的输入是 (E, P)，那么输出是 (E', P') 满足对于任意 $i = 1, ..., k$，如果存在 x 使得 $[a_i]P[x]$，那么 $[a]P'[x]$，且 $E \subset E'$ 的最小指令图。其意义是 a 可以指向 a1 到 ak 可能指向的任何东西。图3.8是这种情形下转移函数的图示。

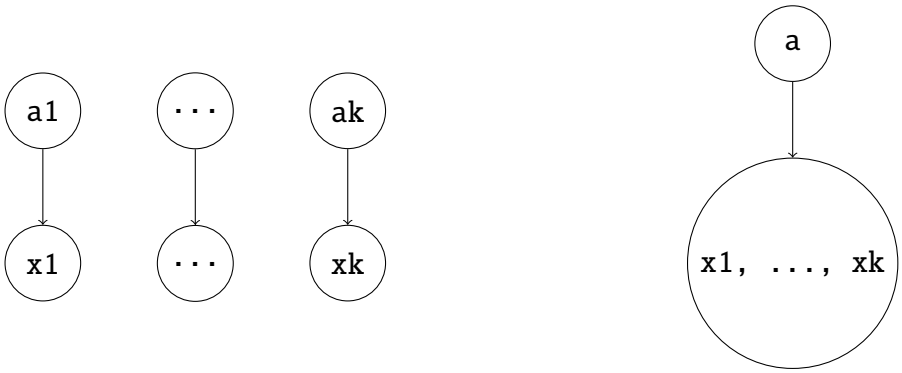


图 3.8　PHINode 转移函数的示例图，左边是输入的指向图，右边是输出的指向图。

对于 CallInst 与 ReturnInst，参数的传递已经在构建定义使用图时通过相应的 PHINode 处理好，但 CallInst 与 ReturnInst 需要处理返回值是一个指针的情况。这种情况下，CallInst 与 ReturnInst 相当于是拷贝指针的 PHINode。所以按 PHINode 的方式处理即可。

3.2.3　增量式优化

在数据流分析在每个结点根据转移函数从输入数据流信息计算输出数据流信息时，如果每次都朴素地按照定义先完整地合并所有前驱输出，再应用转移函数，再和上一次的输出进行比较的方法进行计算，无疑会极大地影响数据流分析的速度。所以我们需要对数据流分析进行增量式优化[30]，也就是说我们对于每个数据流结点维护：它的输入指向图、它的输出指向图、输入指向图相对于上一次的变化（参考定义3）、输出指向图相对于上一次的变化。然后每个结点对应的转移函数也改写为增量形式（定义4），既把输入指向图、输入指向图上的变化的二元组映射为输出指向图上的变化。然后处理每个结点时，先根据前驱的输出指向图的变化更新输入指向图，然后新的输入指向图、输入指向图上的变化、旧的输出指向图计算出输出指向图上的变化，并更新输出指向图。

指向图上的变化量定义如定义3。因为我们的分析是单调的，所以任意结点的输出指向图及输入指向图上的变化总是在半格上变小的变化，因此我们的变化只有两种形式：$[a]$ 和 $[b]$ 合并，以及添加指向关系 $([a], [b])$ 这两种形式，而无需有类似于“拆分”等价关系、“删除”指向关系等形式。

定义 3 指向图上的变化（$\Delta\mathcal{PG}$）是具有这两种形式的数据的集合：

- 合并 $[a]$ 与 $[b]$
- 令 $[a]$ 指向 $[b]$

定义 4 *转移函数的增量形式是一个函数：$(\mathcal{PG}, \Delta\mathcal{PG}) \to \Delta\mathcal{PG}$，把（原始的）输入指向图、输入指向图上的变化映射到输出指向图上的变化。*

不同类型语句的增量形式的转移函数各有不同，其中 StoreInst 较为复杂，其余则相对简单。对于 StoreInst 指令 `store p, a`，其增量形式转移函数定义为：对于输入 $((E, \to), \{d_1, ..., d_k\})$，假设对 $(E, \to)$ 依次应用 $\{d_1, ..., d_k\}$ 之后得到 $(E', \to')$，并假设对 $(E, \to)$ 应用原始转移函数得到的输出指向图为 $(E_o, \to_o)$。如果在 $(E', \to')$ 中 $[p]$ 指向唯一的 x，$[a]$ 指向 $[v]$ 那么该进行 strong-update，输出是 $\{d_1, ..., d_k\}$ 除去“令 $[x]$ 指向 _ ”的变化，再并上“令 $[a]$”指向“$[v]$”，再除去所有在 $(E_o, \to_o)$ 中已经被应用过的变化。如果在 $(E', \to')$ 中 $[p]$ 指向大小大于一的等价类 $[x]$，那么该进行 weak-update，输出是 $\{d_1, ..., d_k\}$，再并上“合并 $[a]$ 与 $[v]$”，再除去所有在 $(E_o, \to_o)$ 中已经被应用过的变化。我们可以证明通过这样的方法计算的每个结点出的输出指向图的变化量的累加值总是和原始计算方法中的输出指向图相等。在实现本段描述的增量式计算时，我们可以始终维护好输入指向图（$(E', \to')$），输出指向图（$(E_o, \to_o)$），而不必重新计算。附录B.1中是本段描述的计算方法的伪代码。

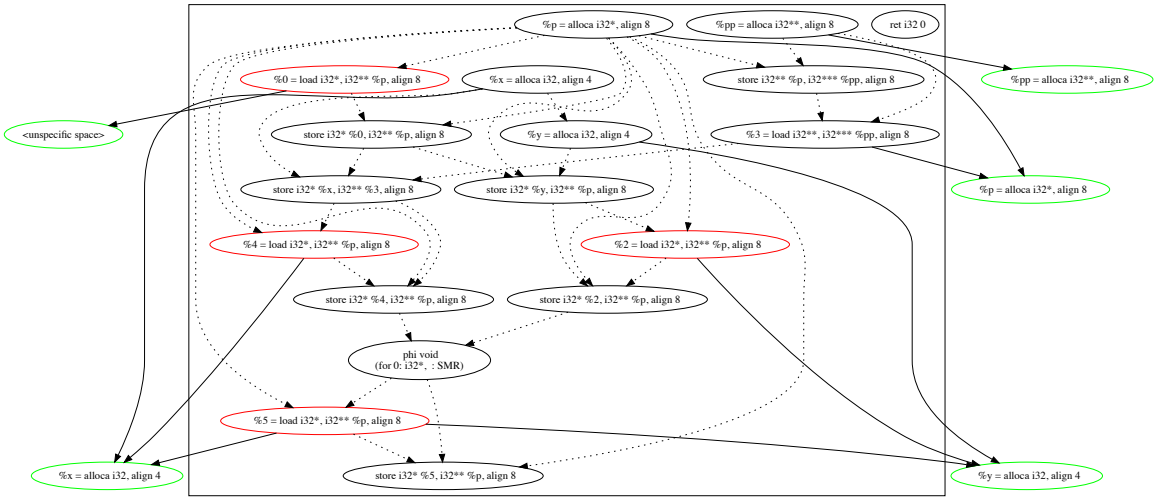
对于 LoadInst 指令 `a = load p`，其增量形式转移函数定义为：对于输入 $((E, \to), \{d_1, ..., d_k\})$，假设对 $(E, \to)$ 依次应用 $\{d_1, ..., d_k\}$ 之后得到 $(E', \to')$，并假设对 $(E, \to)$ 应用原始转移函数得到的输出指向图为 $(E_o, \to_o)$，那么输出是 E' 中 $[p]$ 指向的等价类与 E_o 中 $[p]$ 指向的等价类新增的元素与 E_o 中 $[p]$ 指向的等价类逐个合并。高效计算这一点需要我们的并查集算法支持枚举等价类元素的询问。

对于 `getelementptr` 等指针算术类型的只需合并并传递前驱的输出指向图的变化即可。对于 `phi`、`CallInst` 等合并指针类型的指令除了传递前驱输出指向图的变化还需添加相应的合并各个前驱指向集合的变化量。

接下来我们用一个简单的示例程序来展示我们的算法。图3.9a是一段简单的 C 程序。图3.9b是对应的局部分析阶段的结果。在图3.9b中虚线是局部数据流分析的数据流传递边。黑色实线则连接每条定义一个指针的指令与这条指令返回的指针可能指向的资源。图中四处红色结点分别对应原始代码中四处 `p = p` 指令对 `p` 的读取。可以看出，在分支语句之前的读取 p 的结果是 `<unspecific space>`，在分支中一个分支的结果指向 `x`，另一个分支指向 `y`，而在分支合并之后的读取则既可能指向 `x` 也可能指向 `y`。

```
int *p, **pp, x, y;
pp = &p;
p = p;
if(...) {
  p = &y;
  p = p;
} else {
  *pp = &x;
  p = p;
}
p = p;
return 0;
```

(a) 局部分析示例代码



(b) 局部分析结果。虚线是局部数据流分析的数据流传递边。黑色实线则连接每条定义一个指针的指令与这条指令返回的指针可能指向的资源。图中四处红色结点分别对应原始代码中四处 p = p 指令对 p 的读取。

3.3　自下而上分析阶段

在每个函数调用强连通分量内进行了局部指针分析后，接下来可以进行自底向上计算每个函数的“归纳”，并将被调用者的“归纳”应用在调用者的函数调用处。所以我们遍历每个函数 SCC，对于每个函数调用指令，如果其调用的函数是 SCC 之外的函数，我们首先对其进行递归处理。在被调用者处理完毕之后，本 SCC 内调用的函数只有两种情况：

- 调用一个在本编译单元内没有定义、只有声明的外部函数，比如库函数。
- 调用一个已经有“归纳”的函数。

对于外部函数，简单的处理方法是假设我们总是在进行“全程序分析”（whole-program analysis ），并且对库函数的定义也有“先验知识”，这样在分析中就不会有外部函数。但显然这样的处理方法限制了分析的适用面，所以更好的方法是对外部函数进行恰当的处理。我们的处理方法是，对于外部函数调用 `r = call f(a1, a2, ...)` ，我们将通过`r, a1, a2, ...` 等参数和返回值能够直接或间接指向的内存遍历等价类添加一个“外部标志”，在回答指针分析询问时，任意两个具有外部标志的变量等价类都被视为可能是同一片内存。并且对于通过函数参数能够直接或间接指向的变量等价类，我们也添加这样的外部参数。

对于非外部函数的调用,因为我们已经对其进行了递归处理,所以被调用函数 callee 已经有了“归纳”。所谓一个函数的”归纳“即是指的在这个函数返回时，函数形式参数、返回值所直接或间接指向的内存变量等价类的指向图。由于我们已经在内存 SSA 转换阶段记录了每一个内存变量等价类的最后一次定义，所以我们只需要用最后一次定义位置的输出指向图即可当作“归纳”使用。我们对于被函数调用的归纳的“应用”的方法则是：首先把被调用者的指向图复制一份到调用者的指向图中，然后把调用指令的实际参数指向的变量和被调用者的形式参数所指向的变量合并。

整个自底向上内联算法伪代码如图：

```
void postOrderInline(scc) {
  visited[scc] = true;
  // Step0. process all dependent SCCs first.
  for(all CallInst call in scc) {
    if(called function is not in scc
         && !visited[scc of callee]) {
      postOrderInline(scc of callee);
    }
  }
```

```
  ProcessExternalCalls();

  std::vector<CallInst*> callSitesToProcess;
  for(all CallInst call to functions with summary) {
    callSitesToProcess.push(call);
  }

  while(callSitesToProcess.size() > 0) {
    bool processedOne = false;
    for(each call in callSitesToProcess) {
      if(all arguments for call has been calculated) {
        mergeCallsite(call);
        add successors of calls to worklist;
        chaosIterating();
        remove call from callSitesToProcess;
      }
    }
  }
}
```

注意我们在对于一条函数调用内联被调用者的归纳时，必须要已知这个函数调用的所有实际参数。但由于实际参数可能是其他函数调用的返回值，因此我们必须维护一个待处理函数调用的队列，并且每次从队列中取出可以被处理的函数调用进行内联，然后再重启工作表迭代。

另外，对于通过函数形式参数直接或间接指向的内存我们总是放弃 strong-update，因为我们不知道这些变量是否是唯一的变量。对于这些参数放弃 strong-update 是较为简单的做法。一个可能的优化方法是在被调用者假设所有通过形式参数指向的内存都是唯一的变量（在半格中最大的情况），全部进行 strong-update。而在调用者处应用归纳时根据实际参数的真实情况修正结果。

最后我们用一个例子来结束本节对算法的介绍。图3.10中左图是一个带标注的简单 C 程序，其中的标注 `__print_pointTo(...)` 会被分析程序特殊处理，分析程序会打印出参数所可能指向的集合。右图是分析算法对左图中标注的输出结果，可以看出我们的算法对过程间参数进行了正确的处理，并且没有进行 strong-update。

```
#include "FCPAnnotation.h"

int* g1() {
  static int x;
  return &x;
}

void g2(int **p) {
  static int x;
  *p = &x;
}


int f() {
  int *p = g1();
  __print_pointTo(p);

  g2(&p);
  __print_pointTo(p);
  return 0;
}
```

```
Point-to of
call void @__print_pointTo(i8* %1) {
        @g1.x = internal global i32 0
        <unspecific space>
}

Point-to of
call void @__print_pointTo(i8* %3) {
        <unspecific space>
        @g1.x = internal global i32 0
        @g2.x = internal global i32 0
}
```

图 3.10　自底向上阶段示例。左边是带标注的程序，右边是分析的结果。

3.4　算法复杂度

我们的算法在辅助分析阶段调用DSA算法，故时间复杂度是DSA的复杂度$O(V\alpha(V))$，其中V是程序中语句的条数，$\alpha(\cdot)$是阿克曼函数$A(n,n)$的反函数。在SSA转换阶段进行的是标准的SSA转换算法[7]，一般认为对于实践中的程序的SSA形式以及转换算法的时间复杂度都是线性的。在局部分析阶段，我们进行的是单调数据流分析，因为我们采用了增量形式的实现，故算法的主要时间瓶颈是指向图的变化量的传递。由于我们的指向图是基于合一的，故在每条语句处可能生成的变化量是$O(V)$的，每个变化量都可能沿着本语句相关联的边传播一次，故总的传播次数为$O(VE)$，对于每条信息的处理需要一个并查集上的操作，复杂度为$\alpha(V)$，故总的复杂度为$O(VE\alpha(V))$。自底向上阶段的主要时间瓶颈也是更新数据分析，复杂度也为$O(VE\alpha(V))$。故我们的算法总的时间复杂度是$O(VE\alpha(V))$。相比较而言，SFS算法的主要时间瓶颈和我们类似，不过由于其采用的是基于子集的指向图，故算法的时间复杂度比我们更高，为$O\left(V^2E\right)$。DSA因为在局部采用的是流不敏感算法，故其复杂度比我们更低，为$O(V\alpha(V))$。

我们算法的空间复杂度上的瓶颈则是每个语句处的指向图，在稀疏分析给出最坏结果，即每条语句都需要保存所有的变量的信息的最坏情况下复杂度为$O\left(V^2\right)$。相应

的，SFS 算法的空间复杂度为 $O\left(V^3\right)$，而 DSA 的空间复杂度仍为 $O(V)$。

第四章　实验评估

我们在LLVM框架下对本文描述的CFHS算法进行了初步的实现。实现基于LLVM 3.7，使用 Chris Lattner 等人的 DSA 实现作为辅助分析。我们的实现约有 3000 行 C++ 代码。在本章中我们汇报我们的实现在实际程序集上的准确率，并与 DSA 算法进行比较。

4.1　实验数据集

我们的实验在标准的 SPEC2000 程序集上进行，SPEC 2000 程序集由多个开源 C 语言项目汇总而成，其中包含的程序大小从一千行（art）到 20 万行（gcc），整个程序集一共 60 万行代码。表4.1是对数据集中的程序编译出中间表示后统计的程序特征，表中的数据没有考虑间接函数调用。①。

4.2　实验方法

我们在 SPEC2000 上与 DSA 比较指针分析的精度。本文采用的比较方法是对于程序集中每一个函数，我们对此函数编译产生的中间表示中的所有指针，两两用来询问分析算法这两个指针有没有可能指向同一片内存（may alias）。指针分析算法对于询问给出的回答是如下三种形式之一：

- May Alias，可能指向同样变量
- Not Alias，没有指向同样变量
- Must Alias，一定指向同样变量

然后我们比较不同的指针分析算法能给出的有效回答，即 Not Alias 和 Must Alias 的总数。越精确的分析算法应当给出越多的有效回答。

在之前的研究中，34采用的评估方式是比较指向图结点的平均大小，但这样的比较方法不适用于不基于指向图的别名分析。22采用的评估方法是运行 LLVM 编译器测试集，比较指针分析算法对测试集中别名询问的有效回答数，但此种方法的评估结果受到 LLVM 测试集的影响。故本文选择了对标准程序集中的所有指针进行两两询问的评估方法。

① 数据来自 Sui 等人[34]。

表 4.1　SPEC2000 程序集特征。

Program	KLOC	#Procs	#Pointers	#Loads +Stores	#CallSites	Indirect	#SCCs	Max SCC
164.gzip	8.6	113	3004	586	418	2	0	0
175.vpr	17.8	275	7930	2160	1995	2	0	0
176.gcc	230.4	2256	134380	51543	22353	140	179	398
177.mesa	61.3	1109	44582	17320	3611	671	1	1
179.art	1.2	29	600	103	163	0	0	0
181.mcf	2.5	29	1317	526	82	0	1	1
183.equake	1.5	30	1203	408	215	0	0	0
186.crafty	21.2	112	11883	3307	4046	0	5	2
188.ammp	13.4	182	9829	1636	1201	24	6	2
197.parser	11.4	327	8228	2597	1782	0	42	3
252.eon	41.2	1296	41950	4001	12733	80	17	1
253.perlbmk	87.1	1079	54816	20900	8470	58	12	322
254.gap	71.5	857	61435	22840	5980	1275	33	20
255.vortex	67.3	926	40260	11256	8522	15	12	38
256.bzip2	4.7	77	1672	434	402	0	0	0
300.twolf	20.5	194	20773	8657	2074	0	5	1

4.3　实验结果

表4.2中显示了较大的几个程序上的实验结果以及整个程序集上的实验结果。表中数字为分析算法回答为'May Alias' 的比例（越低越好）。可以看得到本文的 CFSH 算法在大部分情况下比 DSA 能进行更多的有效回答。在 crafty 程序上 CFSH 算法则比 DSA 效果要差，这是因为实验中对 CFSH 的实现是域不敏感的，而 DSA 是域敏感的。但实际上我们的算法也可以同 DSA 一样扩展为域敏感的。

表 4.2　实验结果。CFSH 和 DSA 两行的数字为分析算法回答为'May Alias' 的比例（越低越好）。#Queries 一行为总的询问数。

% of "May."	gcc	perlbmk	gap	mesa	vortex	crafty	**All Repos**
CFSH	36%	31%	11%	7%	12%	19%	**29%**
DSA	53%	39%	27%	42%	12%	11%	44%
#Queries	87035239	16525787	5530823	14970307	5407887	947713	140917215

CFHS 算法相对于 DSA 的一大优势是可以进行 Must Alias 的回答。由于 DSA 是流不敏感的，所以如果两个指针在 DSA 的指向图中指向同一个结点，即使这个结点只有唯一的元素，DSA 也不能断定是否是 Must Alias，其原因是 DSA 的指向图中忽略了"未初始化值"的可能。这是流不敏感算法的一项固有限制，因为任何变量在程序流中的某一个位置总是未初始化的。而 CFSH 算法因为是流敏感的，所以分析结果可以确定地知道某个指针一定指向某个变量而不存在没有初始化的可能。表4.3中列出了 CFSH 算法的有效回答中'Must Alias' 的比例。由于 DSA 相对于我们的实现有域敏感的优势，所以实际上如果我们结合这两个算法的回答的话，可以得到一个更高准确率的分析算法。

表 4.3　CFSH 回答 MustAlias 的比例。

% of "Must."	gcc	perlbmk	gap	mesa	vortex	crafty	**All Repos**
CFSH	8%	7%	14%	45%	13%	11%	13%
#Queries	87035239	16525787	5530823	14970307	5407887	947713	140917215

第五章　结论

本文中我们讨论了指针分析这一经典问题的各类算法在不同维度的精确性。在之前研究的基础上，我们设计了一种流敏感、上下文敏感、堆上下文敏感的指针分析算法 CFHS。我们的算法在过程内进行稀疏的、基于合一的流敏感分析，在过程间进行自底向上内联。在本文作者所知范围内，本文描述的 CFHS 分析算法是第一个流敏感、上下文敏感、堆上下文敏感的指针分析算法。实验结果表明，我们的算法能比流不敏感、域敏感、上下文敏感、堆上下文敏感的 DSA 算法有更高的准确率。

因为指针分析是静态分析领域的基础性问题之一，所以我们的算法能在所有需要指针分析的问题中得到应用。虽然存在之前的研究工作[13] 认为流敏感、上下文敏感的指针分析提供了“不必要的”精确程度。但本文认为实际上存在一些静态分析应用场景，在其中流敏感、上下文敏感的指针分析是必要的。比如内存泄漏自动修复问题[11][38]，在这个问题中，需要设计算法能自动检测出内存泄漏，并且需要找到一个指针能“一定”指向需要被释放的内存来进行释放。因此在这个问题中，指针分析算法一定要能进行“Must Alias”回答，并且对堆进行精确的建模。本文认为在程序自动修复领域对流敏感指针分析具有本质上的需求，在未来工作中探索本文的算法在哪些场景下能有应用是一项值得进行的工作。

对于本文设计的算法来说，未来值得探索的进一步改进方向则包括：

- 利用更先进的数据结构替换并查集或 BDD 来表示指向集的可能性。对于流敏感的分析来说，函数式数据结构[25] 很有可能可以用于改进算法，因为指向图在结点的传递过程中，后继的指向图是和前驱的指向图有重叠的。
- 本文描述的算法在运行时占用时间的主要部分是在数据流图中传递“指向图”上的变化。在 CFHS 中，指向图上的变化是两类基础变化的列表，如果我们能将指向图上的变化表示为相对于列表更紧凑的数据结构，也就是能够将基础变化进行合并，那么也很有可能优化算法的运行效率。在此基础上，我们可以借鉴网络流预流推进类算法对工作表迭代过程的优化[12]，考虑优化工作表的调度顺序，优先进行合并，这样可以减少每个结点在工作表中出现的次数，有可能降低算法复杂度。

参考文献

[1] Lars Ole Andersen. *Program analysis and specialization for the C programming language* [phdthesis], **1994**.

[2] John Aycock and Nigel Horspool. "*Simple generation of static single-assignment form*". In: *International Conference on Compiler Construction*, **2000**: 110–125.

[3] Gianfranco Bilardi and Keshav Pingali. "*Algorithms for computing the static single assignment form*". *Journal of the ACM (JACM)*, **2003**, *50*(3): 375–425.

[4] Jong-Deok Choi, Michael Burke and Paul Carini. "*Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects*". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina, USA: ACM, **1993**: 232–245. http://doi.acm.org/10.1145/158511.158639.

[5] Jong-Deok Choi, Ron Cytron and Jeanne Ferrante. "*Automatic construction of sparse data flow evaluation graphs*". In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, **1991**: 55–66.

[6] Fred C. Chow, Sun Chan, Shin-Ming Liu *et al.* "*Effective Representation of Aliases and Indirect Memory Operations in SSA Form*". In: *Proceedings of the 6th International Conference on Compiler Construction*. London, UK, UK: Springer-Verlag, **1996**: 253–267. http://dl.acm.org/citation.cfm?id=647473.760381.

[7] Ron K Cytron and Jeanne Ferrante. "*Efficiently computing φ-nodes on-the-fly*". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **1995**, *17*(3): 487–506.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen *et al.* "*Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*". *ACM Trans. Program. Lang. Syst.* 1991-10: 451–490. http://doi.acm.org/10.1145/115372.115320.

[9] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore *et al.* "*Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem*". *ACM Trans. Program. Lang. Syst.* 2007-05. http://doi.acm.org/10.1145/1232420.1232424.

[10] Harold N Gabow and Robert Endre Tarjan. "*A linear-time algorithm for a special case of disjoint set union*". In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, **1983**: 246–251.

[11] Qing Gao, Yingfei Xiong, Yaqing Mi *et al.* "*Safe memory-leak fixing for C programs*". *Proceedings - International Conference on Software Engineering*, **2015**, *1*(1): 459–470.

[12] Andrew V Goldberg and Robert E Tarjan. "*A new approach to the maximum-flow problem*". *Journal of the ACM (JACM)*, **1988**, *35*(4): 921–940.

[13] Samuel Z Guyer and Calvin Lin. "*Error checking with client-driven pointer analysis*". *Science of Computer Programming*, **2005**, *58*(1-2): 83–114.

[14] Ben Hardekopf and C. Lin. "*The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code*". *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, **2007**, *42*: 290–299. http://portal.acm.org/citation.cfm?id=1250767.

[15] Ben Hardekopf and Calvin Lin. "*Exploiting pointer and location equivalence to optimize pointer analysis*". In: *International Static Analysis Symposium*, **2007**: 265–280.

[16] Ben Hardekopf and Calvin Lin. "*Flow-sensitive Pointer Analysis for Millions of Lines of Code*". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, **2011**: 289–298. http://dl.acm.org/citation.cfm?id=2190025.2190075.

[17] Ben Hardekopf and Calvin Lin. "*Flow-sensitive pointer analysis for millions of lines of code*". In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, **2011**: 289–298.

[18] Ben Hardekopf and Calvin Lin. "*Semi-sparse Flow-sensitive Pointer Analysis*". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Savannah, GA, USA: ACM, **2009**: 226–238. http://doi.acm.org/10.1145/1480881.1480911.

[19] Vineet Kahlon. "*Bootstrapping: A Technique for Scalable Flow and Context-sensitive Pointer Alias Analysis*". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Tucson, AZ, USA: ACM, **2008**: 249–259. http://doi.acm.org/10.1145/1375581.1375613.

[20] William Landi and Barbara G. Ryder. "*A Safe Approximate Algorithm for Interprocedural Aliasing*". In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. San Francisco, California, USA: ACM, **1992**: 235–248. http://doi.acm.org/10.1145/143095.143137.

[21] Chris Lattner and Vikram Adve. "*LLVM: A compilation framework for lifelong program analysis & transformation*". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, **2004**: 75.

[22] Chris Lattner, Andrew Lenharth and Vikram Adve. "*Making context-sensitive points-to analysis with heap cloning practical for the real world*". *ACM SIGPLAN Notices*, **2007**, *42*: 278.

[23] Erik M. Nystrom, Hong-Seok Kim and Wen-mei W. Hwu; ed. by Roberto Giacobazzi. "*Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis*". In: *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, **2004**: 165–180. http://dx.doi.org/10.1007/978-3-540-27864-1_14.

[24] Erik M Nystrom, Hong-Seok Kim and Wen-mei W Hwu. "*Importance of heap specialization in pointer analysis*". In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, **2004**: 43–48.

[25] Chris Okasaki. "*Functional data structures*". *Advanced Functional Programming*, **1996**: 131–158.

[26] David J. Pearce, Paul H.J. Kelly and Chris Hankin. "*Efficient Field-sensitive Pointer Analysis of C*". *ACM Trans. Program. Lang. Syst.* 2007-11. http://doi.acm.org/10.1145/1290520.1290524.

[27] John H. Reif and Harry R. Lewis. "*Symbolic Evaluation and the Global Value Graph*". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Los Angeles, California: ACM, **1977**: 104–118. http://doi.acm.org/10.1145/512950.512961.

[28] Thomas Reps. "*Program analysis via graph reachability*". *Information and Software Technology*, **1998**, *40*(11-12): 701–726. http://www.sciencedirect.com/science/article/pii/S0950584998000937.

[29] Thomas Reps. "*Undecidability of context-sensitive data-dependence analysis*". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **2000**, *22*(1): 162–186.

[30] Barbara G Ryder and Marvin C Paull. "*Incremental data-flow analysis algorithms*". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **1988**, *10*(1): 1–50.

[31] Micha Sharir and Amir Pnueli. "*Two approaches to interprocedural data flow analysis*". **1978**.

[32] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology Boston, **2006**.

[33] Bjarne Steensgaard. "*Points-to Analysis by Type Inference of Programs with Structures and Unions*". In: *Proceedings of the 6th International Conference on Compiler Construction*. London, UK, UK: Springer-Verlag, **1996**: 136–150. http://dl.acm.org/citation.cfm?id=647473.727458.

[34] Yulei Sui, Sen Ye, Jingling Xue *et al.* "*Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation*". *Software: Practice and Experience*, **2014**, *44*(12): 1485–1510.

[35] William E Weihl. "*Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables*". In: *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, **1980**: 83–94.

[36] John Whaley and Monica S. Lam. "*Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams*". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. Washington DC, USA: ACM, **2004**: 131–144. http://doi.acm.org/10.1145/996841.996859.

[37] John Whaley and Martin Rinard. "*Compositional Pointer and Escape Analysis for Java Programs*". In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. Denver, Colorado, USA: ACM, **1999**: 187–206. http://doi.acm.org/10.1145/320384.320400.

[38] Hua Yan, Yulei Sui, Shiping Chen *et al.* "*Automated memory leak fixing on value-flow slices for C programs*". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, **2016**: 1386–1393.

[39] Jianwen Zhu. "*Towards Scalable Flow and Context Sensitive Pointer Analysis*". In: *Proceedings of the 42Nd Annual Design Automation Conference*. Anaheim, California, USA: ACM, **2005**: 831–836. http://doi.acm.org/10.1145/1065579.1065798.

附录 A　内存 SSA 算法伪代码

本附录为内存 SSA 转换步骤的算法伪代码，算法说明请参考第3.1节。

```
// Step1. place all necessary PHINodes for memory objects
for(each BasicBlock bb in F) {

  for(each Instruction inst in bb) {
    if(inst is a StoreInst) {
      // If a StoreInst modify a memory object aliasing our
      // 'resources', insert a PHINode at the dominance frontier.
      Value* ptr = store->getPointerOperand();

      // Get the pointed variable equivalent class (a DSNode )
      //   from the DSA analysis result
      DSNode *n = dsgraph->getNodeForValue(ptr).getNode();

      for(each DominanceFrontier frontier of bb) {
          Create PhiNode for n at frontier;
      }

    } else if(inst is an AllocInst) {
      // AllocaInst is treated as storing an 'unspecific'
      //   value to the memory.
      Value *ptr = alloca;
      DSNode *n = dsgraph->getNodeForValue(ptr).getNode();

      for(each DominanceFrontier frontier of bb) {
          Create PhiNode for n at frontier;
      }
    } else if(inst is a CallInst){
      ...
    }
  }
}
```

图 A.1　ϕ 放置第一步：对每一条修改变量的指令放置一个 ϕ 指令在支配边界处。

```
// Add PHINode for PHINode inserted earlier
//   until the PHINode set is closed...
queue<BasicBlock*> bbWithNewPHI;
for(each BasicBlock 'bb' with a PHINode) {
  bbWithNewPHI.push(bb);
}
while(!bbWithNewPHI.empty()) {
  BasicBlock* bb = bbWithNewPHI.front();
  bbWithNewPHI.pop();

  for(each PHINode p of bb) {
    DSNode *n = Coresponding DSNode of p;
    for(each DomianceFrontier frontier of bb) {
      if(frontier does not have a PHINode for n) {
        Create PhiNode for n at frontier;
        if(frontier is not in bbWithPHI) {
          bbWithNewPHI.push(frontier);
        }
      }
    }
  }
}
```

图 A.2 ϕ 放置第二步：求 ϕ 结点的闭包

```
void buildSSARenaming(map<DSNode *, Instruction *> &lastDef,
     BasicBlock *bb) {
  // Step1. scan instructions in 'bb', adding new definitions
  //   and linking usage to its last definition.
  lastDefBackup = lastDef;
  for(each PHINode p of bb) {
    DSNode *n = corresponding variable equivalent class of p
    lastDef[n] = p;
  }
  for(each Instruction inst in bb) {
    // Alloca/StoreInst use a value and define a new version of it.
    // LoadInst only use a value.
    if(inst is a StoreInst) {
      Value *ptr = inst->getPointerOperand();
      DSNode *n = dsgraph->getNodeForValue(ptr).getNode();
      Instruction *def = lastDef[n];
      memSSAUsers[def].insert(store);
      // Defines a new
      lastDef[n] = inst;
    } else if(inst is a LoadInst) {
      Value *ptr = inst->getPointerOperand();
      DSNode *n = dsgraph->getNodeForValue(ptr).getNode();
      // Reads an old
      Instruction *def = lastDef[n];
      memSSAUsers[def].insert(inst);
    } else { ... /* deal with other kinds of instructions */}
  // Step2. update PHINodes for successors blocks in the CFG.
  for(BasicBlock *succ : successors(bb)) {
      for(each PHINode p of succ) {
        DSNode *n = corresponding DSNode;
        if(lastDef[n] exists) {
          Instruction *def = lastDef[n];
          memSSAUsers[def].insert(p);
        }
      }
  }
  // Step3. Recursively process immediate dominated BasicBlocks.
  for(each child of bb in DominanceTree)) {
    buildSSARenaming(lastDef, child->getBlock());
  }
  // Step4. Restore definitions added in this BasicBlock.
  lastDef = lastDefBackup;
}
```

图 A.3　ϕ 放置第三步：解析定义使用关系。对于函数调用的处理方式参考图A.4

```
if(inst is ...) {
  ...
} else if(inst is a CallInst) {
  auto& argLastDef = callArgLastDef[inst];
  auto& argRetMerge = callRetMemMergePoints[inst];

  for(each variable equivalent class 'n' modified
        or returned by inst) {
    // Record current 'lastDef' so that we can splice
    //   callsite and callee in
    // the inter-procedural phase.
    if(lastDef[n] exists) {
      Instruction *def = lastDef[n];
      argLastDef[n] = def;
      memSSAUsers[def].insert(argRetMerge[n]);
    } else {
      // 'n' may be returned by the call, so it
      //   may don't have a previous definition.
    }

    // 'lastDef'-s are also connected with the returning
    // merge point so that we can do intra-procedural
    // analysis (function calls are treated as identity functions).
    lastDef[n] = argRetMerge[n];
  }
}
```

图 A.4　图A.3中省略的对函数调用指令的处理方法

附录 B　局部分析阶段算法伪代码

图B.1是局部分析阶段在工作表迭代过程中对 StoreInst 进行增量式计算的伪代码。算法说明请参考小节3.2.3。

```
auto ptr = store->getPointerOperand();
auto content = store->getValueOperand();
auto ptrMem = getMemObjectsForVal(ptr);
auto contentMem = getMemObjectsForVal(content);
outDelta = inDelta;
if(ptrMem && contentMem) {
  if(ptrMem is a unique variable) {  //Strong update
    // Overwrite all previous 'pointTo' modification
    //     of this memory object.
    outDelta.erase_all_elements_with_form( PointsTo(ptrMem, _) );
    auto delta = make_pointTo(ptrMem, contentMem);
    if(!isEmitted(delta, inst)) {
      outDelta.push_back(delta);
      outInDiff.push_back(delta);
    }
  } else { // Weak update.
    auto ptrTo = in.getPointTo(ptrMem);
    if(ptrTo == nullptr
     || !in.eqClass.equivalent(ptrTo, contentMem)) {
      if(ptrTo == nullptr) {
        ptrTo = getImplicitArgOf(ptrMem);
        auto delta = make_pointTo(ptrMem, ptrTo);
        outDelta.push_back(delta);
        outInDiff.push_back(delta);
      }
      auto delta = make_merge(ptrTo, contentMem);
      if(!isEmitted(delta, inst)) {
        outDelta.push_back(delta);
        outInDiff.push_back(delta);
      }
    }
  }
}
```

图 B.1　局部分析阶段增量式计算的 StoreInst 指令伪代码

本科期间的主要工作和成果

本科期间参加的主要科研项目

1. 基于逆向更新的双向程序语言的应用，于 2016 年 6 月至 9 月在日本 National Institute of Informatics 进行，指导老师：胡振江
2. 利用先验知识的词向量训练方法，指导老师：胡俊峰

会议论文：

1. Zhixuan Yang, Chong Ruan, Caihua Li, Junfeng Hu. Optimize Hierarchical Softmax with Word Similarity Knowledge. 17th International Conference on Intelligent Text Processing and Computational Linguistics. Konya, Turkey. April, 2016.

致谢

首先感谢熊英飞老师花费了大量时间指导我完成了这一个毕业设计研究。您在《编程语言设计原理》课上的精彩讲解为我打开了程序语言理论这一优美领域的大门。我在您《软件分析》课程上学得的内容又是本研究的基础。与您前后超过半年的每周讨论是本文的研究能顺利完成的最重要原因。

感谢日本国立情报学研究所的胡振江教授、柯向上博士、朱子润、章雍哲几位师友当我在日本访问交流时对我的照顾，与你们时常进行的线上交流始终充满了乐趣并极大地开拓了我在程序语言领域的眼界。

感谢计算语言所的胡俊峰教授、计算机所的孙薇薇副教授在我本科阶段的初期指导我进行了极有意义的科研训练，并帮助我寻找到了最适合自己的研究方向。从你们学到科研方法、科研态度会让我受益终生。

感谢生活中的各位同学及朋友们，尤其是北京大学茶学社的各位朋友，你们让我对自己的生活感到有趣并在我感到困难时给了我很大的帮助。

最后感谢我的家人，你们永远给我提供了坚定的支持，并且始终明白什么是我想追求的。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在□一年/□两年/□三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日