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设计

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

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[17]，及 Hardekopf 等人的流敏感稀疏分析算法[13]。本文对这两类算法在数据流分析框架下进行一致的理论分析，并在同一理论框架下设计出一个同时流敏感、上下文敏感的指针分析算法。

实验表明本文提出的算法在精度上相较于学界以往成果皆有大幅度改善并同时拥有在实践中可接受的运行效率。在作者所知范围内，这是学界中报告的首个高效流敏感、上下文敏感指针分析算法。

本文同时提出一个用我们的指针分析算法进行内存泄漏自动修复的方法。实验结果表明，我们的算法可以大幅改进学界在此问题上的最先进成果。

关键词：静态分析, 指针分析, 上下文敏感，流敏感

Practical Flow-sensitive and Context-sensitive Points-to Analysis with Heap Cloning

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

ABSTRACT

Test of the English abstract.

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[24] 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）[17]。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[19]，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[10][11][14][12][13]。其最先进的成果 Staged Flow-sensitive Analysis (SFS)[13] 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。在本文描述的分析框架下，流不敏感指针分析总是流敏感指针分析的特例，而基于合一的算法与用 BDD 优化的基于子集的算法都被视作对集合进行压缩编码的一类优化方法的特列。

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 Summary 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。 TODO 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显著提高指针分析的准确率。

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**??**章在单调数据流框架下证明我们算法的正确性，并分析其与相关算法的联系。
- 第**??**章对本文提出的算法进行实验评估。
- 第**??**章指出一些本文提出的分析算法的应用场景。
- 第**??**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[rice] 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[22]。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[Tarjan] 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 Anderson 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 ）[summary] 以及 **CFL 可达性方法**（context-free-language reachability method ）[21] 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，如：**cs-clone...**。

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

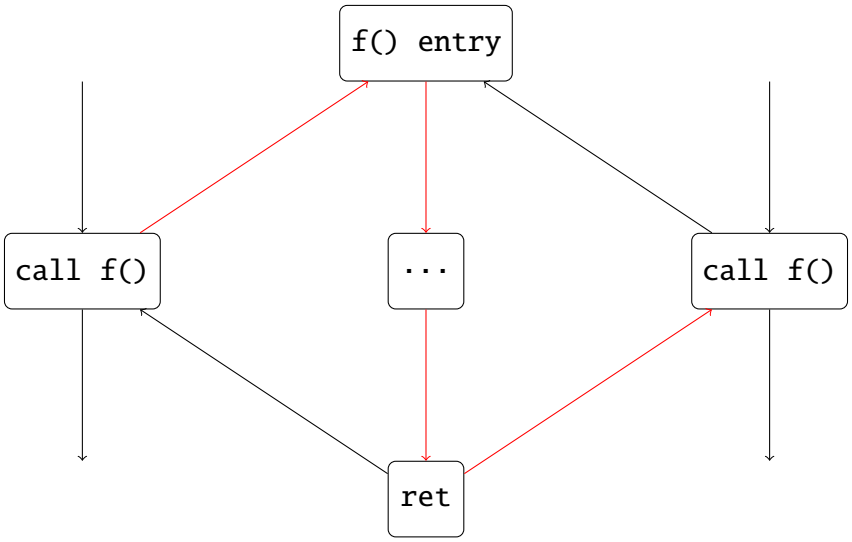


图 2.1　错误的调用－返回匹配路径。上下文不敏感算法允许这样的路径存在。

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[19] 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

```
/* 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

时的整个调用上下文都会被区分。

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[earlyFS...] 没有在实践中得到大规模的使用。Hardekopf 等人在 2009 年[14] 及 2011 年[13] 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[13] 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[repsUndecidable] 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[fieldSensitivity] 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。

	基于合一	基于子集	流敏感
上下文不敏感	• Weihl 1980[24] 第一篇指针分析的研究工作 • Steensgaard 1996[22] 第一个高效的分析算法	• Andersen 1994[1] • Hardekopf 2007[10] 进行动态 SCC 检测优化	• Choi 1993[4] • Hardekopf 2009[14] 2011[12] 基于子集
上下文敏感	• Lattner 2007[17] 且是域敏感、堆上下文敏感的	• Whaley 2004[25] • Nystrom 2004[18] 且是堆上下文敏感的 • Sui 2014[23]	• Zhu 2005[26] • Kahlon 2008[15] • **本文的算法** 且是堆上下文敏感的

图 2.3　指针分析算法汇总

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[20]。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

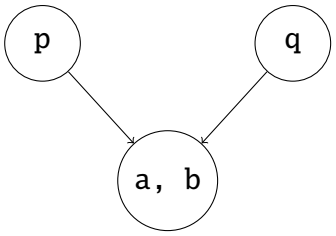
指针的存在使得 SSA 转换变得困难，所以现代编译器的中间表示格式，比如 LLVM IR[16]，往往是“部分 SSA 形式”，即被取过地址的变量仍然是可修改的内存中的变量形式，而没有被取过地址的变量则可以被优化为静态单赋值形式。而如果我们想把程序中的所有变量转换为 SSA 形式，则必须要进行指针分析，从而知道通过指针进行的间接访问修改的是哪些变量。但因为指针分析生成的是“保守”的结果（即指针分析结果认为 p 指向 a，实际上 p 不一定真的会指向 a，但指针分析一定包含真实的结果），所以通过指针分析识别出的间接访问、修改实际只是“潜在“间接修改。我们也采用 Chow 等人[6] 的方法，对于间接写入（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[12] 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（12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。

3.1.3 过程间内存 SSA

3.2 局部分析阶段

3.3 自下而上分析阶段

结论

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

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] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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 B　附件

致谢

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