第 7 章

指针分析与优化

指针是很多编程语言包含的重要特性，它本质上抽象了内存地址并允许程序通过它来间接访问内存单元。多个指针变量可能指向同一内存位置，这种现象被称为*别名*（aliasing）。别名的存在使得程序分析需要考虑多个指针变量对同一内存位置的潜在影响，从而使得程序分析和优化更加复杂和困难。指针分析是为了有效应对这种复杂性，而进行的一类重要的静态程序分析，其目标是通过分析指针之间的别名关系，来确定哪些指针可能指向相同的内存位置。指针分析不但可以揭示潜在的代码优化机会，还为分析程序安全性分析奠定了基础。

7.1 分配位置抽象与指向分析

指针本质上是对程序内存的一种间接引用，由于它混淆了程序中的数据流信息，给程序分析和优化都带来了挑战。例如，如下示例程序中的 x、y 是整型指针变量，而 z 是整型变量：

```
1  *x = 42;
```

```
2 *y = 0;
3 z = *x;
```

在零值分析中，如果指针 x、y 指向不同的内存地址，则可以确定变量 z 的值肯定是非零值（实际上是 42）；而如果指针 x、y 指向相同的内存地址，则可以确定变量 z 的值肯定是零值。而在无法静态确定指针 x、y 的指向关系的情况下，则我们也无法得到精确的零值分析结果。同理，在常量传播优化中，如果我们无法静态确定指针 x、y 的指向关系，则也无法判定是否能够将常量的值传播给变量（如本例中的 z 的值）。一般的，如果两个指针变量（如上述的 x、y）指向同一个内存单元，我们称其为**别名**（Aliasing）。因此，我们需要对程序进行指针分析，以确定其中的指针变量的别名关系，进而基于该关系，进行更精确的程序分析和优化。

我们首先对第??小节中的控制流图定义进行扩展，加入对指针操作的支持。具体来说，我们对表达式 E 的产生式做如下扩展（其它部分保持不变）：

$$E ::= \ldots \mid \&x \mid *x \mid alloc(x) \mid assign(x, y)$$

其中，$\&x$ 表示取变量 x 的地址，$*x$ 表示对指针变量 x 解引用取其内容，$alloc(x)$ 表示在内存中分配 x 字节的空间并返回分配得到的地址，而 $assign(x, y)$ 表示把变量 y 的值，写入指针 x 所指向的内存空间中。为简单起见，我们假设内存分配 $alloc(x)$ 总是成功，且经常省略其参数，而将其简写为 $alloc()$。并且，我们经常将指针写入操作 $assign(x, y)$ 写为 $*x = y$，以便和许多语言的语法形式表示保持一致。

指针的**指向分析**（points-to analysis）的目标是通过静态分析，确定每个指针变量 x 在运行时可能指向的内存单元的

集合。然而，由于程序每次运行可能会分配不同甚至是无穷的内存单元，因此，直接将所有的动态可分配的内存单元作为指针分析的指向对象集合非常困难。为解决这一问题，我们可以采用**分配位置抽象**（allocation-site abstraction），它将程序中的变量名称以及分配操作所在的位置，作为抽象内存单元。直观上，该方法将每个程序变量可能的指向单元、以及分配操作可能分配的内存单元，都分别视为不同的等价类。

具体来说，对每个程序变量 x，我们引入一个与变量同名的抽象单元 x，表示该变量动态可能指向的内存单元集合。同理，对于每个动态内存分配操作 $alloc$，我们引入一个唯一标识的抽象单元 $alloc_i$，其中 i 是唯一的索引（如该分配操作所在的唯一代码行号），用以代表在相应代码位置分配的所有实际内存单元。这样，给定程序中所有抽象单元的集合是确定且有限的，我们用 $Cells$ 来表示该集合。

基于分配位置抽象，指向分析的结果是一个映射

$$[\![\,]\!] : x \to \mathcal{P}(Cells),$$

将指针变量 x 映射到其可能指向的分配位置抽象集合 $[\![x]\!] \in \mathcal{P}(Cells)$。指向分析的结果是对运行时信息的保守估计，即对变量 x 的分析结果集合 $[\![x]\!]$ 是运行时实际指向内存单元集合的超集，这意味着分析结果 $[\![x]\!]$ 不会遗漏任何实际运行时 x 可能指向的内存单元，但可能包含变量 x 实际运行中不会指向的内存单元。

例 7.1 *对代码*

```
1 p = alloc();
2 q = &p;
3 r = &q;
```

我们可得到抽象单元集合

$$Cells = \{p, q, r, alloc_1\}。$$

而我们对该程序进行指向分析，可能得到的一种结果是

$$\begin{aligned}[\![p]\!] &= \{alloc_1\} \\ [\![q]\!] &= \{p\} \\ [\![r]\!] &= \{q\}\,,\end{aligned}$$

即指针变量 p 可能指向分配位置抽象集合 $\{alloc_1\}$，而指针变量 q、r 可能分别指向分配位置抽象集合 $\{p\}$ 和 $\{q\}$。

基于指向分析的结果，我们可以进一步推导指针间的其它重要关系。例如，基于指向信息可以得到指针间的别名关系：即指针变量 x 和 y 是别名，当且仅当

$$[\![x]\!] \cap [\![y]\!] \neq \emptyset。$$

在本章余下章节中，我们将首先讨论进行指向分析的 Andersen 算法和 Steensgaard 算法。随后，我们将讨论指针分析的扩展，包括用于跨函数推导指针关系的过程间指针分析，以及用于确定控制流依赖性的流敏感指针分析。最后，我们将介绍指针分析的实际应用，包括用于发现潜在运行时错误的空指针检测，以及用于优化内存管理逃逸分析等。

7.2 Andersen 算法

Andersen 算法是一种基于集合包含（inclusion-based）关系的指向分析，它为每个变量维护一个指向集合，并通过计算集合间的包含关系，得到指向集合的值。

表 7.1: Andersen 算法的约束生成规则。

指针操作	约束
$x = alloc()$	$alloc_i \in [\![x]\!]$
$x_1 = \&x_2$	$x_2 \in [\![x_1]\!]$
$x_1 = x_2$	$[\![x_2]\!] \subseteq [\![x_1]\!]$
$x_1 = *x_2$	$c \in [\![x_2]\!] \Rightarrow [\![c]\!] \subseteq [\![x_1]\!]$， 对$\forall c \in Cells$
$*x_1 = x_2$	$c \in [\![x_1]\!] \Rightarrow [\![x_2]\!] \subseteq [\![c]\!]$， 对$\forall c \in Cells$
$x = null$	/

Andersen 算法主要分成两步：首先，分析算法扫描目标程序，对每个变量 x，生成一组针对指向集合 $[\![x]\!]$ 间的约束，约束表达为 $[\![x]\!]$ 集合间的包含关系。其次，分析算法对生成的约束进行求解，得到每个变量 x 的指向集合 $[\![x]\!]$。

7.2.1 约束生成

表7.1给出了 Andersen 算法对待分析程序的不同语法，生成对应约束的规则。具体而言，对于指针分配语句 $x = alloc()$（分配的内存的大小这里不重要），算法将抽象内存单元 $alloc_i$ 加入到变量 x 的指向集合 $[\![x]\!]$ 中。对于取地址语句 $x_1 = \&x_2$，算法将抽象内存单元 x_2 加入到变量 x_1 的指向集合 $[\![x_1]\!]$ 中。对于指针赋值语句 $x_1 = x_2$，算法将变量 x_2 的指向集合 $[\![x_2]\!]$ 加入到变量 x_1 的指向集合 $[\![x_1]\!]$ 中。对于指针解引用 $x_1 = *x_2$，算法将变量 x_2 集合中的每个元素 c 所对应的指向集合 $[\![c]\!]$，加入到变量 x_1 的指向集合 $[\![x_1]\!]$。对于指针指向内存的赋值 $*x_1 = x_2$，算法对变量 x_1 指向集合 $[\![x_1]\!]$ 中的每个抽象内存单元 c，都将变量 x_2 的指向集合 $[\![x_2]\!]$ 加入到 c 的执行集合 $[\![c]\!]$ 中。最后，空指针赋值 $x = null$ 不改变任何指向集合。

对整个程序，Andersen 算法依次扫描程序中的语句一遍，

最终生成约束的集合。由于 Andersen 算法不考虑语句的执行流关系，因此属于流不敏感分析 （Flow-insensitive analysis）。

例 7.2 对如下示例程序，Andersen 算法生成的约束如右侧注释中所示，其中 $Cells = \{p, q, x, y, z, alloc_1\}$。

```
1 p = alloc(); // alloc_1 ∈ [[p]]
2 x = y; // [[y]] ⊆ [[x]]
3 x = z; // [[z]] ⊆ [[x]]
4 *p = z; // c ∈ [[p]] ⇒ [[z]] ⊆ [[c]], ∀c ∈ Cells
5 p = q; // [[q]] ⊆ [[p]]
6 q = &y; // y ∈ [[q]]
7 x = *p; // c ∈ [[p]] ⇒ [[c]] ⊆ [[x]], ∀c ∈ Cells
8 p = &z; // z ∈ [[p]]
```

最终，Andersen 算法为该程序生成的约束集合为

$$\begin{aligned}\{&alloc_1 \in [\![p]\!],\ [\![y]\!] \subseteq [\![x]\!],\ [\![z]\!] \subseteq [\![x]\!],\\ &c \in [\![p]\!] \Rightarrow [\![z]\!] \subseteq [\![c]\!], [\![q]\!] \subseteq [\![p]\!], y \in [\![q]\!],\\ &c \in [\![p]\!] \Rightarrow [\![c]\!] \subseteq [\![x]\!], z \in [\![p]\!]\}。\end{aligned}$$

简单起见，我们省略了其中的量词约束 $\forall c$。

需要注意的是：Andersen 算法的流不敏感性，决定了语句扫描的先后顺序不影响最终的约束集合。例如，即便我们任意交换例7.2 中语句的先后顺序，Andersen 算法仍然会生成相同的约束集合。

7.2.2 约束求解

Andersen 算法需要求解生成的集合约束，以得到每个变量 x 的指向集合 $[\![x]\!]$。注意到：每个变量 x 的指向集合 $[\![x]\!]$ 都单调增大，且存在上界 $Cells$，因此我们可以使用算法 7.1给出的迭代算法。算法接受程序 P 作为输入，计算并返回每个变

算法 7.1 基于迭代的 Andersen 算法

输入： 程序 P

输出： 变量的指向集合 $[\![\cdot]\!]$

```
1: function Andersen(P)
2:     Cons = generateConstraints(P)                    ▷ 根据表7.1
3:     for each x ∈ Cells do
4:         [[x]] = ∅
5:     while any point-to set [[x]] still changes do
6:         for each constraint C ∈ Cons do
7:             switch C do
8:                 case alloc_i ∈ [[x]]:
9:                     addToSet({alloc_i}, [[x]])
10:                case x_2 ∈ [[x_1]]:
11:                    addToSet({x_2}, [[x_1]])
12:                case [[x_2]] ⊆ [[x_1]]:
13:                    addToSet([[x_2]], [[x_1]])
14:                case c ∈ [[x_2]] ⇒ [[c]] ⊆ [[x_1]] for each c ∈ Cells:
15:                    for each variable c ∈ [[x_2]] do
16:                        addToSet([[c]], [[x_1]])
```

量 x 的指向集合 $[\![x]\!]$。算法首先按照表7.1中的规则，计算得到程序 P 的所有约束形成的集合 $Cons$。接着，算法开始求解约束 $Cons$ 集合。为此，算法首先将所有变量 x 的指向集合 $[\![x]\!]$ 初始化为空集 $\emptyset$，然后开始迭代。在每次迭代过程中，算法逐个对约束集合 $Cons$ 中的每条约束 C 根据其语法形式，进行分情况讨论，并调用 addToSet(s, t) 函数将集合 s 加入到集合 t 中。整个算法一直迭代到所有变量 x 的指向集合 $[\![x]\!]$ 都不再变化，即达到不动点后，执行终止。

对例7.2中的示例，应用 Andersen 算法，得到所有变量 x 的指向集合 $[\![x]\!]$ 的迭代过程如表 7.2所示。最终，算法在迭代三次后终止，得到的各个变量的指向集合如表7.2最后一列所

表 7.2: Andersen 算法计算变量的指向集合的迭代过程。

指向集合	初始化	迭代 1	迭代 2	迭代 3
$[\![p]\!]$	$\emptyset$	$\{alloc_1, z\}$	$\{alloc_1, z, y\}$	$\{alloc_1, z, y\}$
$[\![q]\!]$	$\emptyset$	$\{y\}$	$\{y\}$	$\{y\}$
$[\![x]\!]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[\![y]\!]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[\![z]\!]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

示。

7.2.3 约束高效求解

Andersen 算法的执行过程涉及求解一系列集合约束，而这类问题的求解通常具有较高的计算成本。因此，研究如何高效地求解此类约束至关重要。本节讨论一种通用的**立方体算法**（Cubic algorithm），它能够在 $O(n^3)$ 时间内计算得到最小解，适用于包括 Andersen 算法求解在内的很多集合约束求解问题。

我们考虑一个通用的子集约束求解问题，其结构与 Andersen 算法中的指针分析约束类似。具体地，我们有标记集 $T = \{t_1, \ldots, t_k\}$，对应 Andersen 算法中的指向目标集合；和变量集 $X = \{x_1, \ldots, x_n\}$，其中每个变量 x 的取值 $[\![x]\!] \in \mathcal{P}(T)$ 是一个有限的标记集合。在不引起混淆的情况下，我们下面常把 $[\![x]\!]$ 简记为 x。

约束包括两类：

1. 直接包含约束：$t \in x$ ，即某个标记 t 直接属于变量 x 的解集；

2. 条件包含约束：$t \in x \Rightarrow y \subseteq z$，即如果 t 属于 x，则变量 y 的解集必须包含于 z 的解集。

为了有效求解该约束系统，我们维护一个有向图 $G = (V, E)$，其中节点 V 对应变量集合 X，边表示节点间的包含关系（即变量之间的子集关系），亦即对于 $\langle x, y\rangle \in E$，我们有 $x \subseteq y$。

对于每个变量 x，我们定义辅助数据结构：

1. 解集：$x.sol \subseteq T$，存储当前已知属于 x 的标记集合；
2. 后继集合：$x.succ \subseteq V$，存储所有必须包含 x 取值的变量（即与 x 之间的包含关系）；
3. 条件约束：$x.cond(t) \subseteq V \times V$，表示与 x 及标记 t 相关的条件约束。

此外，我们使用一个工作列表 $W \subseteq T \times V$ 来存储待处理的标记-变量对。在初始化阶段，所有集合为空，随后逐一处理约束，并使用算法 7.2中定义的辅助函数来添加标记、添加边、以及传播标记。

对于形如 $t \in x$ 的约束，直接将标记 t 加入变量 x 的解集中，并触发一次传播过程，确保所有已经出现的约束都得到满足。即：

```
addToken(t, x)
propagate()
```

而对于形如 $t \in x \Rightarrow y \subseteq z$ 的条件包含约束，如果 t 已经在 x 的解集中，则立即添加包含关系 $y \subseteq z$ 并再次进行传播；否则，该约束被暂存，等待 t 进入 x 的解集时再执行。即：

算法 7.2 Andersen 算法约束求解辅助函数

```
1:  procedure addToken(t, x)
2:      if t ∉ x.sol then
3:          x.sol.add(t)
4:          W.add(t, x)
5:  procedure addEdge(x, y)
6:      if x ≠ y and y ∉ x.succ then
7:          x.succ.add(y)
8:          for t ∈ x.sol do
9:              addToken(t, y)
10: procedure propagate()
11:     while W ≠ ∅ do
12:         (t, x) = W.removeNext()
13:         for y ∈ x.succ do
14:             addToken(t, y)
```

```
if t ∈ x.sol then
  addEdge(y, z)
  propagate()
else
  x.cond(t).add(y, z)
```

该算法的时间复杂度可以通过分析操作次数得出。假设程序的规模为 n，即变量和标记的数量均为 $O(n)$，而约束的数量为 $O(n^2)$。在求解过程中，每个标记最多被传播 $O(n^2)$ 次，而每次传播的计算量为 $O(n)$，因此总复杂度为 $O(n^3)$。这一复杂度在 Andersen 算法的指针分析中是难以突破的，因此被称为立方时间瓶颈。

尽管该方法已经是当前已知的最优精确求解算法，但 $O(n^3)$ 的复杂度在大型程序上仍然可能过高。因此，研究者们提出了一系列优化策略，例如循环消除（合并存在循环依赖的变量）、按拓扑顺序处理工作列表（减少重复计算）、交错执行传播和约束处理（提高计算效率）、差分传播（仅传播

发生变化的部分），以及按需处理（减少不必要的计算）。

7.2.4 字段敏感分析

如果待分析的程序中使用了记录，则记录的字段也可能是指针。为处理指针字段，我们可以使用一类称为**字段敏感**（Field sensitive）的分析方法，即跟踪每个字段可能的指向集合。

为此，我们将指向集合 $[\![\cdot]\!]$ 扩展为

$$[\![\cdot]\!] : (Cells \cup (Cells \times Fields)) \to \mathcal{P}(Cells),$$

其中 $Fields = \{f_1, \ldots, f_n\}$ 是被分析程序中出现的字段名的集合。为了和源语言的记号保持一致，我们将二元组 (c, f) 记为符号 $c.f$，表示访问变量 c 的字段 f，其中 $c \in Cells, f \in Fields$。

基于这些符号，我们在表7.3中给出了对不同语句的约束生成规则。对于记录的整体赋值语句 $x = \{f_1 : x_1, \ldots, f_k : x_k\}$，生成的约束将每个指针类型字段 x_i，$1 \le i \le k$，的指向集合 $[\![x_i]\!]$ 都加入到对应的二元组指向集合 $[\![x.f_i]\!]$ 中。对于记录分配语句 $x = alloc\{f_1 : x_1, \ldots, f_k : x_k\}$，生成的约束将抽象内存单元 $alloc_i$ 加入变量 x 的指向集合 $[\![x]\!]$，并将每个变量 x_j，$1 \le j \le k$，的指向集合 $[\![x_j]\!]$ 分别加入到字段 f_j 的指向集合 $[\![alloc_i.f_j]\!]$ 中。对于记录字段的读 $x_1 = x_2.f$，生成的约束将字段 $x_2.f$ 的指向集合 $[\![x_2.f]\!]$，加入到变量 x_1 的指向集合 $[\![x_1]\!]$ 中。对于指针赋值 $x_1 = x_2$，生成的约束将指针 x_2 的指向集合 $[\![x_2]\!]$ 加入到指针 x_1 的指向集合 $[\![x_1]\!]$ 中，并将指针 x_2 的每个字段 $x_2.f$ 的指向集合 $[\![x_2.f]\!]$，都加入到指针 x_1 的对应字段 $x_1.f$ 的指向集合 $[\![x_1.f]\!]$ 中。对于指针的内存读语句 $x_1 = *x_2$，

表 7.3: 字段敏感的指针操作及约束

指针操作	约束
$x = \{f_1 : x_1, \ldots, f_k : x_k\}$	$[\![x_1]\!] \subseteq [\![x.f_1]\!] \wedge \ldots \wedge [\![x_k]\!] \subseteq [\![x.f_k]\!]$
$x = alloc\,\{f_1 : x_1, \ldots, f_k : x_k\}$	$alloc_i \in [\![x]\!] \wedge [\![x_1]\!] \subseteq [\![alloc_i.f_1]\!] \wedge \ldots \wedge [\![x_k]\!] \subseteq [\![alloc_i.f_k]\!]$
$x_1 = x_2.f$	$[\![x_2.f]\!] \subseteq [\![x_1]\!]$
$x_1.f = x_2$	$[\![x_2]\!] \subseteq [\![x_1.f]\!]$
$x_1 = x_2$	$[\![x_2]\!] \subseteq [\![x_1]\!] \wedge [\![x_2.f]\!] \subseteq [\![x_1.f]\!]$ for each $f \in$ Fields
$x_1 = *x_2$	$c \in [\![x_2]\!] \Rightarrow ([\![c]\!] \subseteq [\![x_1]\!] \wedge [\![c.f]\!] \subseteq [\![x_1.f]\!])$ for each $c \in Cells$ and $f \in Fields$
$*x_1 = x_2$	$c \in [\![x_1]\!] \Rightarrow ([\![x_2]\!] \subseteq [\![c]\!] \wedge [\![x_2.f]\!] \subseteq [\![c.f]\!])$ for each $c \in Cells$ and $f \in Fields$

生成的约束将变量 x_2 的指向集合 $[\![x_2]\!]$ 中的每个元素 c 所对应的指向集合 $[\![c]\!]$ 都加入到变量 x_1 的指向集合中 $[\![x_1]\!]$，并将 c 每个字段的指向集合 $[\![c.f]\!]$ 都加入到变量 x_1 对应字段的指向集合 $[\![x_1.f]\!]$ 中。对于指针的内存写语句 $*x_1 = x_2$，生成的约束对变量 x_1 的指向集合 $[\![x_1]\!]$ 中的每个元素 c，将变量 x_2 的指向集合 $[\![x_2]\!]$ 加入 c 的指向集合 $[\![c]\!]$ 中，并将 x_2 每个字段的指向集合 $[\![x_2.f]\!]$ 都加入到 c 对应字段的指向集合 $[\![c.f]\!]$ 中。

例 7.3 对如下程序，其 $Cells = \{x, y, z, i, t\}$ 及 $field = \{f, h\}$，我们应用表7.3中的规则，得到右侧的约束。

```
1 i = null; // ∅
2 x = alloc {f: i}; // alloc1 ∈ [[x]] ∧ [[i]] ⊆ [[alloc1.f]]
3 y = alloc {h: x}; // alloc2 ∈ [[y]] ∧ [[x]] ⊆ [[alloc2.h]]
4 t = *y; // c ∈ [[y]] ⇒ ([[c]] ⊆ [[t]] ∧ [[c.f]] ⊆ [[t.f]]对c ∈ Cells且f ∈ Fields
5 z = t.h; // [[t.h]] ⊆ [[z]]
```

最终，我们得到程序的这些指向集为 $[\![x]\!] = [\![z]\!] = \{alloc_1\}$，$[\![y]\!] = \{alloc_2\}, [\![i]\!] = [\![t]\!] = \emptyset$。

最后，我们要指出的是，求解包含字段的约束，我们仍然可以采用前面讨论的迭代算法或立方体算法。

7.3 Steensgaard 算法

Steensgaard 算法是一种基于变量等价（equivalence-based）关系的指向分析，它为每个变量维护一个抽象变量，并通过计算抽象变量间的等价关系，得到指向集合的值。Steensgaard 算法主要步骤分成两步：首先，分析算法扫描程序，并生成一组针对每个抽象变量 x 间的约束，约束表达为变量等价关系；其次，分析算法对生成的约束进行求解，得到每个变量 x 的指向集合 $[\![x]\!]$。

7.3.1 约束生成

Steensgaard 算法在约束生成过程中，会产生符合如下文法的项

$$t ::= [\![c]\!] \mid \alpha \mid \uparrow t$$

其中 $[\![c]\!]$ 是对每个抽象单元 $c \in Cells$ 引入的抽象变量，但和 Andersen 算法不同，这里的 $[\![c]\!]$ 仅表示一个待求解的项变量，并不直接表示抽象内存单元构成的集合。变量 α 是约束生成过程中生成的临时项变量。而一元构造符 $\uparrow t$ 表示指向项 t 的指针。

我们在表7.4中给出了在 Steensgaard 算法的约束生成规则。对于指针分配语句 $x = alloc()$，算法令变量 x 的抽象变量 $[\![x]\!]$ 和 $alloc_i$ 对应的抽象变量 $\uparrow [\![alloc_i]\!]$ 相等。对于取地址语句 $x_1 = \&x_2$，算法令变量 x_1 的抽象变量 $[\![x_1]\!]$ 和变量 x_2 的

表 7.4: Steensgaard 算法的指针操作及约束

指针操作	约束
$x = alloc()$	$[\![x]\!] =\uparrow [\![alloc_i]\!]$
$x_1 = \&x_2$	$[\![x_1]\!] =\uparrow [\![x_2]\!]$
$x_1 = x_2$	$[\![x_1]\!] = [\![x_2]\!]$
$x_1 = *x_2$	$[\![x_2]\!] =\uparrow \alpha \wedge [\![x_1]\!] = \alpha$，其中 α 是一个新变量
$*x_1 = x_2$	$[\![x_1]\!] =\uparrow \alpha \wedge [\![x_2]\!] = \alpha$，其中 α 是一个新变量

项 $\uparrow [\![x_2]\!]$ 相等。对于指针赋值语句 $x_1 = x_2$，算法令变量 x_1 和 x_2 的抽象变量 $[\![x_1]\!]$ 和 $[\![x_2]\!]$ 相等。对于指针解引用 $x_1 = *x_2$，算法新引入一个新的项变量 α，并令变量 x_2 的抽象变量 $[\![x_2]\!]$ 等于项 $\uparrow \alpha$，且令变量 x_1 的抽象变量 $[\![x_1]\!]$ 等于变量 α。对于指针指向内存的赋值 $*x_1 = x_2$，算法新引入一个新的项变量 α，并令变量 x_1 的抽象变量 $[\![x_1]\!]$ 等于项 $\uparrow \alpha$，且令变量 x_2 的抽象变量 $[\![x_2]\!]$ 等于变量 α。

注意，表中的约束生成规则为每个指针操作最多构造两个约束。对比 Andersen 算法的约束生成规则，其对内存的读写运算，会对每个抽象单元格都有一个约束。因此，Steensgaard 算法在空间复杂度方面相比 Andersen 算法更低。

从表中的约束可以看到，每个变量 x 的抽象变量 $[\![x]\!]$ 表示该变量的一种可能的取值（即指向），因此，该变量 x 最终的指向集合为

$$\{t \in Cells \mid [\![x]\!] =\uparrow [\![t]\!]\}。$$

例 7.4 对于7.2节中的示例程序 7.2，Steensgaard 算法生成如右侧所示的约束。

```
1 p = alloc(); // [[p]] =↑ [[alloc_1]]
2 x = y; // [[x]] = [[y]]
3 x = z; // [[x]] = [[z]]
```

```
4 *p = z; // [[p]] =↑ α1, [[z]] = α1
5 p = q; // [[p]] = [[q]]
6 q = &y; // [[q]] =↑ [[y]]
7 x = *p; // [[x]] = α2, [[p]] =↑ α2
8 p = &z; // [[p]] =↑ [[z]]
```

7.3.2 约束求解

Steensgaard 算法基于术语统一，统一规则遵循以下公理：

$$\uparrow \alpha_1 = \uparrow \alpha_2 \Rightarrow \alpha_1 = \alpha_2$$

我们可以使用**合一**（Unification）算法来找到结果指向集合：

$$pt(p) = pt(q) = \{alloc_1, y, z\}$$

这个结果不如我们使用 Andersen 算法得到的精确，但是使用的算法更快。

7.3.3 约束高效求解

并查集（union-find）数据结构是由一个有向图组成，每个节点有一个指向父节点的边（如果节点是其自己的父节点，那么它是根节点）。如果两个节点有相同的祖先节点，那么这两个节点是等价的，根节点即为它们的代表元素。在并查集中，我们主要有以下三个操作，对应的伪代码如算法7.3所示：（1）MakeSet(x)：创建一个新节点 x，并将其父节点初始化为自己。（2）Find(x)：查找节点 x 的根节点（即它的标准代表）。在查找过程中，会进行路径压缩，使得路径上经过的每个节点的父节点都直接指向根节点，从而加速后续的查找操作。（3）Union(x, y)：将节点 x 和节点 y 所在的集合合并。如

算法 7.3 并查集的三种操作

```
1: procedure MakeSet(x)
2:     x.parent = x
3: procedure Find(x)
4:     if x.parent ≠ x then
5:         x.parent = Find(x.parent)
       return x.parent
6: procedure Union(x, y)
7:     x^r = Find(x)
8:     y^r = Find(y)
9:     if x^r ≠ y^r then
10:        x^r.parent = y^r
```

果它们已经在同一个集合中，则不做任何操作。合并时，会选择其中一个节点作为根节点。

在 Steensgaard 算法中，统一过程依赖并查集来高效地解决约束问题。我们将每个类型术语，包括子术语（例如，在术语 $\uparrow\tau$ 中，我们称 τ 为子术语）看作并查集中的一个节点，并使用 MakeSet(τ) 为每个术语初始化一个集合。此时，每个术语要么是类型变量，要么是具体类型（如整数、指针或函数）。

对于每个约束 $\tau_1 = \tau_2$，我们调用算法7.4中的函数 Unify(τ_1, τ_2) 来统一这两个术语。若它们可以统一，就通过递归统一它们的子术语，从而实现类型的统一。如果尝试统一两个构造函数不同的术语（例如，函数类型具有不同的参数个数），统一会失败。

需要注意的是，Union(x, y) 操作具有不对称性：它总是将第二个参数 y 的根节点作为合并后的标准表示。这意味着，只有在两个术语都是类型变量时，我们才会考虑将一个类型变量统一为另一个类型变量，而对于具体类型，统一时它们将优先成为标准表示。

算法 7.4 Steensgaard 算法约束求解辅助函数

```
1:  procedure Unify(τ1, τ2)
2:      τ1^r ← Find(τ1)
3:      τ2^r ← Find(τ2)
4:      if τ1^r ≠ τ2^r then
5:          if τ1^r and τ2^r are both type variables then
6:              Union(τ1^r, τ2^r)
7:          else if τ1^r is a type variable and τ2^r is a proper type then
8:              Union(τ1^r, τ2^r)
9:          else if τ1^r is a proper type and τ2^r is a type variable then
10:             Union(τ2^r, τ1^r)
11:         else if τ1^r and τ2^r are proper types with the same type constructor then
12:             Union(τ1^r, τ2^r)
13:             for each pair of sub-terms τ1' and τ2' of τ1^r and τ2^r, respectively do
14:                 Unify(τ1', τ2')
15:         else
16:             unification failure
```

当所有约束都处理完之后，获取解变得非常简单。对于每个程序变量或表达式，只需调用 Find() 查找其标准表示。如果该标准表示有子术语，我们递归地为每个子术语查找解。唯一的特殊情况是，当递归导致出现无限类型时，我们会引入 μ 术语来表示这种无限递归。

Steensgaard 算法的优点在于，它只需处理每个约束一次。因此，在实际的实现中，我们可以将约束的生成和求解过程交替进行。也就是说，在生成约束的同时，实时求解这些约束，而不是先生成所有约束再进行统一求解。这使得 Steensgaard 算法在类型分析中能够高效处理约束问题，并在近乎线性时间内得到结果。

7.4 指针分析的扩展

可以对基本的指针分析算法进行扩展，以处理更丰富的语言特性或提高其分析精度。本小节，我们讨论对指针分析的两种扩展：过程间指针分析和流敏感指针分析。

7.4.1 过程间指针分析

在支持函数指针特性的语言中，函数调用

$$x = x_0(x_1, \ldots, x_n); \tag{7.1}$$

中的函数指针 x_0 可能指向多个目标函数，因此过程间指针分析和控制流分析产生了相互依赖问题，即过程间指针分析需要控制流分析确定 x_0 可能调用的目标函数，而控制流分析又需要指针分析提供变量 x_0 的可能指向集合。为了解决这一问题，一种可行的做法是同时执行控制流分析和指针分析。

Andersen 算法已经具备一些控制流分析的特性，因此我们可以通过扩展 Andersen 算法的约束生成规则来实现联合分析。具体地，对于函数指针变量（如上述的 x_0），我们用符号 $[\![x_0]\!]$ 表示其可能指向的具体目标函数集合，其计算规则和 Andersen 算法一致。则对于函数调用7.1，我们可以生成约束

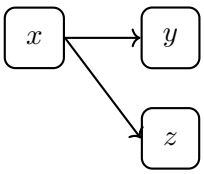
$$f \in [\![x_0]\!] \Rightarrow ([\![x_1]\!] \subseteq [\![y_1]\!] \wedge \ldots \wedge [\![x_n]\!] \subseteq [\![y_n]\!] \wedge [\![y]\!] \subseteq [\![x]\!])$$

其中，函数 f 是有 n 个指针参数的函数定义

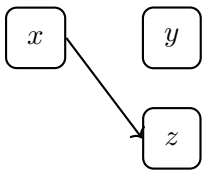
$$f(y_1, \ldots, y_n)\,\{\ldots; \texttt{return}\ y;\}$$

```
*y = 42;
*z = 0;
x = y;
x = z;
print(*x);
```

(a) 待分析的指针程序。



(b) 流不敏感分析得到的结果。



(c) 流敏感分析得到的结果。

图 7.1: 对示例程序进行流不敏感和流敏感分析的结果。

7.4.2 流敏感指针分析

我们已经讨论的指针分析算法，没有考虑程序的执行流，因此称为**流不敏感分析**（Flow-insensitive analysis）。尽管流不敏感分析比较高效，但由于其不考虑程序的执行流，因此得到的分析可能可能相对粗糙。考虑图7.1a中的待分析指针程序（假设变量 y 和 z 不是别名），我们对其进行指针分析可得变量 x 的指向集合

$$[\![x]\!] = \{y, z\}\ ,$$

指针间的指向关系如图7.1b所示。由于变量 x 有多个可能指向，导致我们无法对 `print` 语句中的指针解引用 $*x$ 进行常量传播优化，将其替换为常量。而如果我们考虑语句实际的执行流，则执行流到达 `print` 语句时，变量 x 实际只能指向变量 z，即 $[\![x]\!] = \{z\}$，其指向关系如图7.1c所示。显然，基于该指向关系，我们可以进行常量传播优化，将常量 0 传播到 `print` 语句中。

为了得到更精确的分析结果，我们需要建立一种考虑程序执行流的**流敏感分析**（Flow-sensitive analysis）方法。为此，我们将扩展 Andersen 算法，建立一种基于前向执行流的分析算法。算法使用**指向图**（Point-to graph）来刻画语句的指向状

态，指向图 $G = (V, E)$ 是有向图，其节点 $V = Cells$，即是给定程序的抽象内存单元集合。而边

$$E = \{(v_1, v_2) \mid v_2 \in [\![v_1]\!]\},$$

即指针 v_1 指向 v_2。给定两个指向图 $G_1 = (V_1, E_1)$ 和 $G_2 = (V_2, E_2)$，我们称图 G_2 是图 G_1 的子图，如果

$$V_2 = V_1 \quad \text{且} \quad E_2 \subseteq E_1。$$

例如，考虑图7.1中的两个指向图，显然图7.1c是图 7.1b的子图。

我们用指向图构成的幂集格来刻画数据流关系，格的偏序关系是指向图的子集关系。特别的，底元 $\perp$ 是没有边的图，而顶元 $\top$ 是完全图。

对于每个控制流图的节点 n，我们用 $In[n]$ 和 $Out[n]$ 分别表示该节点前后的指向图。基于对语句 n 语法形式的归纳，给出如下的数据流方程：

$$\begin{aligned}
x = alloc() &: Out[n] = In[n] \downarrow x \cup \{(x, alloc_i)\} \\
x_1 = \&x_2 &: Out[n] = In[n] \downarrow x_1 \cup \{(x_1, x_2)\} \\
x_1 = x_2 &: Out[n] = assign(In[n], x_1, x_2) \\
x_1 = *x_2 &: Out[n] = load(In[n], x_1, x_2) \\
*x_1 = x_2 &: Out[n] = store(In[n], x_1, x_2) \\
x = \texttt{null} &: Out[n] = In[n] \downarrow x
\end{aligned}$$

对于汇合点 v，我们有

$$In[v] = \bigsqcup_{w \in pred(v)} Out[w],$$

其中算符 $\sqcup$ 通过将图的有向边 E 合并，将多个指向图合并为一个（最小）指向图。

上述数据流方程使用了如下记号（其中 G 代表指向图）：

$$
\begin{aligned}
G \downarrow x &= \{(s,t) \in \sigma \mid s \neq x\} \\
assign(G,x,y) &= G \downarrow x \cup \{(x,t) \mid (y,t) \in G\} \\
load(G,x,y) &= G \downarrow x \cup \{(x,t) \mid (y,s) \in G \wedge (s,t) \in G\} \\
store(\sigma,x,y) &= G \cup \{(s,t) \mid (x,s) \in G \wedge (y,t) \in G\}
\end{aligned}
$$

需要注意的是，在最后对存储操作 $store$ 的规则中，由于赋值目标的指向集合 $[\![*x_1]\!]$ 可能是多集，因此，我们无法确定赋值究竟修改了多集中的哪些元素。为此，我们使用*弱更新*（Weak updating），直接将相关的有向边加入到指向图 G。

例 7.5 *考虑图7.1中的程序，应用流敏感的前向数据流分析后，得到的指向图如图*7.1c *所示。*

例 7.6 *考虑图*7.2a *中给出的指针程序，应用数据流方程，得到的指向图如图* 7.2b *所示，该指向图由两个各包含* 4 *个节点的不连通子图构成。*

7.5 指针分析的应用

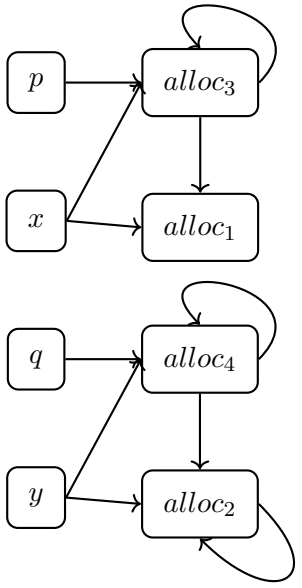
指针分析广泛应用在程序分析、编译优化和软件安全等很多领域。本小节，我们讨论指针分析在软件安全领域的两个典型应用：空指针和悬空指针检测。

7.5.1 空指针检测

在指针解引用 $*x$ 中，如果指针 $x = null$，则我们称其为空指针引用。空指针引用可能引起内存错误或抛出空指针异常。因此，我们希望能够对空指针引用进行静态检测。

```
1 var x, y, n, p, q;
2 x = alloc(); y = alloc();
3 *x = null; *y = y;
4 n = input();
5 while(n>0){
6    p = alloc(); q = alloc();
7    *p = x; *q = y;
8    x = p; y = q;
9    n = n-1;
10}
```

(a) 待分析的指针程序。



(b) 数据流分析得到的指向图。

图 7.2: 对示例程序进行流敏感得到的指向图。

为检测空指针，我们使用如下的包含两个节点的格 N：

$$\begin{array}{c} \top \\ | \\ \bot \end{array}$$

其中底元 $\bot$ 表示指针肯定不为空，而顶元 $\top$ 表示指针可能为空。为了跟踪变量的抽象状态，我们使用如下的映射格：

$$States = Cells \to N$$

将每个抽象内存单元 $Cell$，映射为格 N。

对于控制流图的每个节点 n，我们分别用 $In[n]$ 和 $Out[n]$，分别表示节点 n 前后的抽象状态。我们基于对节点 n 语法形式的归纳，给出前向数据流方程的定义：

$$x = alloc() : Out[n] = In[n][x \mapsto \bot, alloc_i \mapsto \top]$$

$$
\begin{aligned}
x_1 = \&x_2 &: Out[n] = In[n][x_1 \mapsto \bot] \\
x_1 = x_2 &: Out[n] = In[n][x_1 \mapsto In[n](x_2)] \\
x_1 = *x_2 &: Out[n] = load(In[n], x_1, x_2) \\
*x_1 = x_2 &: Out[n] = store(In[n], x_1, x_2) \\
x = \texttt{null} &: Out[n] = In[n][x \mapsto \top]
\end{aligned}
$$

其中

$$
load(In[n], x_1, x_2) = In[n]\left[x_1 \mapsto \bigsqcup_{\alpha \in [\![x_2]\!]} In[n](\alpha)\right]
$$

$$
store(In[n], x_1, x_2) = In[n]\left[\alpha_{\alpha \in [\![x_1]\!]} \mapsto In[n](\alpha) \sqcup In[n](x_2)\right]
$$

亦即对指针加载语句 $x_1 = *x_2$，我们需要考虑指针 x_2 的指向集合 $[\![x_2]\!]$ 中所有的指针。而对于存储操作 $*x_1 = x_2$，由于 $[\![x_1]\!]$ 可能包含多个抽象内存单元，而即便单个抽象内存单元也可能表示多个实际内存单元，因此我们需要使用弱更新，将 $[\![x_1]\!]$ 中每个抽象内存单元 α，都更新为其原始值 $In[n][\alpha]$ 和变量 x_2 值 $In[n][x_2]$ 的最小上界。

对于数据流的汇合点 v，我们有

$$
In[v] = \bigsqcup_{w \in pred(v)} Out[w]。
$$

基于空指针分析的结果，在程序点 n 上的指针解引用 $*x$ 是安全的，如果条件

$$
In[n](x) = \bot
$$

成立。

例 7.7 考虑以下示例程序：

```
1 p = alloc(); // [p ↦ ⊥, q ↦ ⊥, n ↦ ⊥, alloc1 ↦ ⊤]
2 q = &p; // [p ↦ ⊥, q ↦ ⊥, n ↦ ⊥, alloc1 ↦ ⊤]
3 n = null; // [p ↦ ⊥, q ↦ ⊥, n ↦ ⊤, alloc1 ↦ ⊤]
4 *q = n; // [p ↦ ⊤, q ↦ ⊥, n ↦ ⊤, alloc1 ↦ ⊤]
5 *p = n; // [p ↦ ⊤, q ↦ ⊥, n ↦ ⊤, alloc1 ↦ ⊤]
```

Andersen 算法计算的指向集合如下:

$$[\![p]\!] = \{alloc_1\}\ ,\ [\![q]\!] = \{p\}\ ,\ [\![n]\!] = \emptyset$$

则应用前向数据流方程，可得到每条语句后的抽象状态，如上述程序右侧的注释所示。

7.5.2 逃逸分析

如果一个函数将自身局部变量的地址作为指针返回，则我们称该指针发生了逃逸（Escape）。例如，在如下示例程序中，foo 函数返回的局部变量 x 的指针 $\&x$ 发生了逃逸。由于函数返回后，包含局部变量的函数栈帧被销毁，因此对逃逸指针的访问可能导致悬空指针引用（Dangling pointer reference）的内存错误。例如，如下的悬空指针引用 $*p$ 将可能覆盖任意的值。

```
1 int *foo(){
2   int x;
3   return &x;
4 }
5 int main(){
6   int *p;
7   p = foo();
8   *p = 1;
9   return *p;
10 }
```

我们可以基于指针分析的结果，进行逃逸分析（Escape

analysis）来捕获悬空指针引用错误。具体来说，我们首先对函数进行指针分析，获得每个指针 x 的指向集 $[\![x]\!]$。然后，对于返回语句 `return` x，我们只需要检查函数本身定义的参数或变量是否属于 $[\![x]\!]$ 即可。

7.6 本章小结

指针分析是静态分析中的关键技术，在编译器优化和程序分析中发挥重要作用。本章首先介绍了指针分析的基本概念，并简述了别名分析和分配位置抽象的方法。在此基础上，本章详细探讨了两种经典的指向分析技术，即基于集合包含关系的 Andersen 算法和基于变量等价关系的 Steensgaard 算法，并分析了它们在精度与效率上的差异。随后，本章进一步扩展到更高级的指针分析技术，包括过程间指针分析和流敏感指针分析，以提高分析能力和适用范围。最后，本章讨论了指针分析在空指针检测和逃逸分析中的应用。

7.7 深入阅读

指针分析是编译器优化和静态分析中的重要研究方向，已有大量研究探讨了指向分析的精度、效率及其在程序分析中的应用。

1990 年，Chase 等人 [3] 提出了分配位置抽象，为后续的指向分析奠定了基础。随后，Andersen[1] 于 1994 年提出了 Andersen 算法，这是一种基于集合包含关系的指向分析方法，能够提供精确的分析结果，但由于其依赖子集约束求解，计算复杂度较高。Blackshear 等人 [2] 进一步探讨了 Andersen 算法的精度限制，并指出其流敏感性、路径敏感性及过程间敏

感性才是影响指向分析实际效果的关键因素。

为提升指向分析的效率，Steensgaard[4] 提出了一种替代方案，即 Steensgaard 算法，其基于等价类合并，能够在近乎线性的时间复杂度内完成分析，但牺牲了部分精度。

7.8 思考题

1. 使用 Andersen 算法计算以下程序片段中变量的指向集合：

```
1  z = &x;
2  w = &a;
3  a = 42;
4  if (a > b) {
5    *z = &a;
6    y = &b;
7  } else {
8    x = &b;
9    y = w;
10 }
```

2. 使用 Andersen 算法计算以下程序片段中变量的指向集合:

```
a = &b;
a = &x;
b = &c;
c = &d;
x = &y;
y = &z;
*a = **a;
```

注意，最后一条语句没有被规范化。正如 Horwitz[Hor97] 所观察到的，语句的规范化会导致以下意义上的不精确: 根据 Andersen 的分析，这需要使用两个临时变量对语句进行规范化，b 和 x 可能是别名，但对于包含上述赋值集的任何 TIP 程序来说，这是不可能的，无论它们之间存在哪种控制流。

3.

```
a = &c;
c = &b;
b = &a;
b = a;
*b = c;
d = *a;
```

1. 说明 Andersen 算法对这段代码的分析得出 d 可能指向 c 的结论。

2. 对于任何具有这组赋值的 TIP 程序，无论它们之间存在哪个控制流，d 在任何执行中都不会指向 c。(提示: a 可以指向 b, b 可以指向 c，但不能同时指向。)

4. 如果一个分析的所有约束函数都是分配的，那么它就是分配的。说明 Andersen 算法不是分配的。(提示: 考虑语句 x = *y 或 *y = x 的约束。)

5. 使用 Andersen 算法对以下程序进行字段敏感的指向分析。

```
1 x = alloc {f: 1, g: 2};
2 y = alloc {h: x, g: 3};
3 z = alloc 4;
4 y.h = z;
```

6. 使用 Steensgaard 算法计算练习 1. 中程序的指向集合，并分析其结果与 Andersen 分析相比在精度上的差异。

7. 对如下程序分别进行流不敏感和流敏感的指向分析，并比较其结果。

```
1 z = &x;
2 w = &a;
3 a = 42;
4 if(a > b) {
```

```
5   *z = &a;
6   y = &b;
7 } else {
8   x = &b;
9   y = w;
10 }
```

参考文献

[1] Lars Ole Andersen. Program analysis and specialization for the c programming language. 1994.

[2] Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. The flow-insensitive precision of andersen's analysis in practice. In Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18, pages 60 – 76. Springer, 2011.

[3] David R Chase, Mark Wegman, and F Kenneth Zadeck. Analysis of pointers and structures. ACM SIGPLAN Notices, 25(6):296 – 310, 1990.

[4] Bjarne Steensgaard. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 32 – 41, 1996.