第32卷 第9期 计 算 机 学 报 Vol. 32 No. 9
2009年9月 CHINESE JOURNAL OF COMPUTERS Sept. 2009

激进域敏感基于合并的指针分析

于洪涛 张兆庆
(中国科学院计算技术研究所计算机系统结构重点实验室 北京 100190)

摘 要 指针分析是静态程序分析的基础,指针分析的精度直接影响后续的程序分析和优化.域敏感性用来描述指针分析是否需要区分结构体对象的不同域成员.文中提出一种激进的基于合并的域敏感指针分析方法,利用目标机器模型中的数据布局信息进行高层分析,使用基地址和偏移的组合来激进地表示一个结构体域成员以能更精确地区分结构体的不同域成员.文中还对原有类型推导规则做了重要改进,尽量避免在合并类型变量时造成的精度损失.为了保证新类型推导规则的正确性,方法将所有的结构体赋值操作转换成对每个结构体成员的赋值操作.大量实验数据表明,该方法分析精度显著高于以往方法而运行开销几乎相当.该方法还将域成员的激进表示集成至编译器的中间表示中以获得可移植性.

关键词 域敏感的;基于合并的;Steensgaard风格;指针分析;别名分析
中图法分类号 TP311 **DOI号**:10.3724/SP.J.1016.2009.01722

An Aggressively Field-Sensitive Unification-Based Pointer Analysis

YU Hong-Tao ZHANG Zhao-Qing
(*Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

Abstract Pointer analysis is the basis of most other static program analyses for C programming language. The precision of pointer analysis is crucial to optimizing compilers and software productivity tools. Field-sensitivity is used to describe whether a pointer analysis needs to distinguish different field members. Field-insensitive pointer analysis considers all fields of one structural object as the same object. On the contrary, field-sensitive pointer analysis considers different fields as different objects. This paper proposes an aggressively field-sensitive unification-based pointer analysis. Different from existed methods, the method takes target machine architecture into consider in the phase of high-level analysis in order to precisely distinguish fields of structure objects. In the method, a field of a structural object is aggressively represented by a pair of offset from its base structure and size of its own data type. The original inference system is improved to avoid the loss of precision due to joining type variables. All structural memory operations are flattened to a series of scalar memory operations based on the target machine information to guarantee the correctness of type inference system. Lots of experiments indicate that the new method is more precise than the existed method while maintaining almost the same efficiency. Furthermore, the method is portable since the aggressive field representation have been implemented on the intermediate representation of the authors' compiler.

Keywords field-sensitive; unification-based; Steensgaard-style; pointer analysis; alias analysis

---

收稿日期:2009-04-14;最终修改稿收到日期:2009-07-21.本课题得到国家"八六三"高技术研究发展计划项目基金"以编译技术为支撑的高可靠软件开发工具与环境"(2008AA01Z115)资助.**于洪涛**,男,1981年生,博士研究生,主要研究方向为程序分析及编译优化.E-mail:htyu@ict.ac.cn.**张兆庆**,女,1938年生,研究员,博士生导师,主要研究领域为先进编译技术及相关工具环境.E-mail:zqzhang@ict.ac.cn.

1 引　言

指针是一种特殊的数据类型，指针变量用来保存某个程序对象的地址，允许我们能够间接地操作这个程序对象. 在C语言中，指针的使用非常广泛. 一个原因是，指针常常是表达某个程序对象的唯一途径，比如动态分配的存储对象. 另一个原因是，使用指针可以生成更高效的代码，比如将指针作为函数参数可以传递数组或大型结构体对象.

在C语言程序分析中，指针分析用来分析一个指针所有可能指向的存储位置，包括程序中的全局变量、局部变量、动态数据区域等. 指针分析是一类特殊的数据流问题，这个问题的解决程度如何将直接影响其它很多数据流问题解决得好坏. 因此对于含有指针别名行为的程序，指针分析是静态程序分析的基础. 在过去的20年里，指针分析问题的研究已经成为程序分析技术研究的重点之一，至今依然是一个活跃的研究领域. 指针分析的研究内容主要是在精度和时空效率之间做取舍. 指针分析的精度可以用某个程序点指针指向集合的大小来衡量. 分析得出的指针指向集合越大说明该指针分析的精确度越低. 精确的指针分析可以提高优化编译器生成代码的运行速度，有时加速比高达25%以上[1]. 不仅是优化编译器，性能良好的程序检错器也同样依赖于指针分析的精度. 精确的指针分析可以减少检错器的错误误报及漏报，提高检错准确率.

在过去的二十年里，指针分析研究领域涌现了大量的优秀成果. 其中最具影响力的两个成果是Andersen的工作[2]和Steensgaard的工作[3]. Andersen的工作被称为Andersen风格的或者基于包含(inclusion-based)的指针分析，是一种流不敏感的指针分析，具有三次方的多项式时间复杂度. Steensgaard的工作被称为Steensgaard风格的或者基于合并(unification-based)的指针分析. Steensgaard风格的分析也是一种流不敏感的指针分析，优点是具有接近线性的时间复杂度，但是精度低于Andersen风格的方法. 之后指针分析领域内很多研究工作都是在改进Andersen和Steensgaard的工作，如改进Andersen风格的分析的有Hardekopf等的工作[4-6]；改进Steensgaard风格的分析的有Steensgaard、Das等的工作[7-9].

域敏感性用来描述指针分析是否需要区分结构体对象的不同域成员. 域不敏感的分析将同一结构体对象的所有域成员看作一个存储对象. 相反的，域敏感的分析将程序中所有结构体对象的每个成员都视作不同存储对象参与分析. 例如C代码片段

```
struct foo {int *p; int *q;} s1;
int x, y;
s1.p=&x;
s1.q=&y;
```

域不敏感的指针分析将得到 $s1$ 指向 x 和 y，也就是 $s1.p$，$s1.q$ 都可能会指向 x 和 y 这样的结果. 而域敏感的分析将能得到 $s1.p$ 指向 x，$s1.q$ 指向 y. 在C语言中使用结构体对象是很普遍的，从测试集SPEC2000/2006中可以看出，对结构体成员的访存操作占全体访存操作的百分比，最少是1.2%，最多达29.2%，平均是13.7%，因此域敏感的指针分析是十分必要的.

本文将介绍作者实现的一种域敏感的基于合并的指针分析方法. 不同于以往的方法[7]，本方法将目标机器模型中的数据布局信息应用于高层分析之中，并对文献[7]中的推导规则做了重要改进，改进了合并函数及类型体系以避免精度损失，并能对数组元素建模. 大量实验数据表明，本方法分析精度显著高于文献[7]，而运行开销几乎相当.

2 研究动机

Steensgaard本人对其著名工作[3]进行了改进，形成域敏感版本[7]. 文献[7]中采用了一种最大公共子前缀的技术来建立结构体类型之间的对应关系. 我们借用文献[7]中的一个例子，如图1.

Steensgaard风格分析基于等价类的特性会导致很多对象的合并. 图1(a)给出一个例子说明这种情况. 对 $i2$ 的两次赋值导致 $s2$ 和 $s4$ 所指向的对象合并，即 $s1$ 和 $s3$ 需要被合并成一个新的对象，也就是说 $s1$ 和 $s3$ 是别名的. 由于不考虑目标机器模型中的数据布局信息，文献[7]无法知道结构体域成员 $s1.c$ 和 $s3.f$ 的类型长度是否相同，因此在合并 $s1$ 和 $s3$ 时，$s1.c$，$s3.f$ 和 $s3.g$ 都被合并成新结构体对象的一个域成员. 分析得到的结果指向图如图2(b).

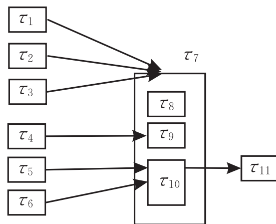
Yong[10]将Steensgaard的思想进一步扩展，给出了3种与目标机器无关的方式解析结构体类型之间的对应关系，并将这些方式推广到各种指针分析，其中最精确的方式与文献[7]相当，都是最大公共子前缀思想. Yong[10]还给出了一种利用目标机器信息进行域敏感分析的模式，然而他并没有讨论如何

利用目标机器信息改进 Steensgaard 风格的分析．因为这种改进并不是一种平凡的改进，需要改进算法本身．此外再没有工作来改进 Steensgaard 风格分析的域敏感性．

Yong[10]认为利用目标机器信息的缺陷是指针分析工具将不可移植．然而在现代编译器里我们可以从机器模型中得到数据布局信息，并尽早地反映在中间表示上．事实上编译器 Open64 已经为我们提供了这样的基础，展示了方法的可移植性．本文将要给出的激进分析会得到图 2 所示的指向图．

```
int   i1, *i2, **i3, **i4;
float  f1, **f2;
struct {int a, *b, *c;} s1, *s2;
struct {int d, *e; float f, *g} s3, *s4;
s2=&s1;
s4=&s3;
f2=&s4->g;
*f2=&f1;
i3=&s2->b;
i4=&s2->c;
*i4=&i1;
i2=(int*) s2;
i2=(int*) s4;
```

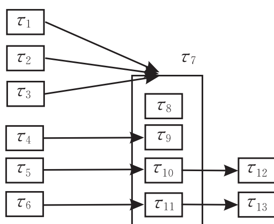
（a）一个 C 程序



i2:	τ_1	*s1*:	τ_7	*s1.c*:	τ_{10}
s2:	τ_2	*s3*:	τ_7	*s3.f*:	τ_{10}
s4:	τ_3	*s1.a*:	τ_8	*s3.g*:	τ_{10}
i3:	τ_4	*s3.d*:	τ_8	*i1*:	τ_{11}
i4:	τ_5	*s1.b*:	τ_9	*f1*:	τ_{11}
f2:	τ_6	*s3.e*:	τ_9		

（b）文献[7]中分析得到的指向图

图 1　域敏感的 Steensgaard 风格分析[7]



i2:	τ_1	*s1*:	τ_7	*s1.c*:	τ_{10}
s2:	τ_2	*s3*:	τ_7	*s3.f*:	τ_{10}
s4:	τ_3	*s1.a*:	τ_8	*s3.g*:	τ_{11}
i3:	τ_4	*s3.d*:	τ_8	*i1*:	τ_{12}
i4:	τ_5	*s1.b*:	τ_9	*f1*:	τ_{13}
f2:	τ_6	*s3.e*:	τ_9		

图 2　针对图 1(a)的例子，激进的分析在目标机器为 IA32 的情况下得到的指向图

3　中间表示

我们先定义程序中可能出现的赋值语句的抽象语法，如图 3，不难看出所有 C 程序的赋值语句和函数调用语句都能转换成我们定义的语法．

$S ::= MEM_1 = MEM_2$	＃存储对象
$\quad \mid MEM = MEM_2$	＃取地址
$\quad \mid MEM = allocate(c)$	＃存储分配
$\quad \mid MEM = op(MEM_1 \cdots MEM_n)$	＃表达式操作
$\quad \mid MEM = fun(f_1 \cdots f_n) \rightarrow r$	＃函数指针赋值
$\quad \mid MEM = CALL(MEM_1 \cdots MEM_n)$	＃函数调用
$MEM ::= x \mid {}^*MEM \mid MEM_1[MEM_2]$	
$\qquad \mid MEM.\langle offset, size \rangle$	
$CALL ::= f \mid {}^*MEM$	

图 3　赋值语句的抽象语法

C 语言规范不允许直接对结构体变量进行强制类型转换，因此可以利用目标机器模型中的数据布局信息将程序中所有结构体变量之间的赋值展开为对所有成员变量的赋值．如果某个成员是结构体则继续展开，直到程序中所有的赋值语句都是在内定义数据类型之间发生．展开看起来似乎会增加代码体积，然而这种展开是必然的，即使不在中间表示时展开，在生成代码时一样会展开，所以它并不会影响代码长度．到后面会看到，这种展开非常有利于我们进行类型推导．

4　类型系统

和文献[7]中方法一样，首先我们建立一套非标准类型系统．类型系统由类型和类型规则组成．类型系统是基于合并的指针分析的基础，它规定了指针分析应该具有的性质并且保证了指针分析的正确性．非标准类型系统中的类型并不是标准的数据类型，而是描述存储对象的指向情况．类型规则用来保证程序是类型良好的(well-typed)．一个程序是类型良好的，是指静态指针分析的结果可以保守地描述程序实际运行时的动态存储构造[3]．

为程序中的每个存储对象赋予一个非标准类型变量，用来描述该存储对象的指向情况．顺序处理每条程序语句，对这些类型变量进行类型推导得到最终的类型环境．最后利用类型环境构建指向图，这就是整个指针分析过程．因此还要针对每条类型规则设定相应的推导规则以保证在处理每条程序语句时类型规则都成立．类型推导过程实际上就是指针分

析的实施过程,而推导规则就相当于具体的分析算法.本节我们先介绍类型和类型规则,第 5 节再具体介绍推导规则.

4.1 类　型

类型用来描述存储对象的指向情况.为程序中的每个存储对象赋予一个非标准类型变量,用来描述该存储对象的指向情况.

定义 1. 定义如下非标准类型

$$\alpha ::= \tau \times \lambda$$
$$\tau ::= \bot \mid \text{blank} \mid \text{simple}(\alpha \times size) \mid \text{struct}(m) \mid \text{object}(\alpha)$$
$$\lambda ::= \bot \mid \text{lam}(\tau_1 \cdots \tau_n)(\tau_{n+1})$$
$$m ::= \bot \mid (\langle offset, size \rangle \to \tau)\ \text{mapping}$$

α 类型用来描述值,或者是存储对象,或者是函数对象.为每个存储对象赋予一个 τ 类型的类型变量.存储对象包括程序中出现的标量、数组、结构体以及堆对象.τ 类型变量的值(或者称为类型)表示如下4 种情况:

(1) blank.该存储对象是什么类型目前未知;

(2) simple($\alpha \times size$).该存储对象是一个标量,α 描述它指向的存储对象或函数,$size$ 指出它的类型长度;

(3) struct(m).该存储对象是一个结构体对象.m 是一个映射关系,映射该结构体对象所有可能的域成员到域成员的类型变量.我们用二元组〈偏移,类型长度〉表示结构体对象的一个域成员.

(4) object(α).该存储对象的存储布局我们无法得知,α 描述它指向的存储对象或函数.

这 4 种情况为每个函数对象赋予一个 λ 类型的类型变量.λ 类型变量的值记载了该函数的形参和返回值信息.

对图 1(a)的例子,经类型推导之后所有对象的类型变量如图 4 所示,其指向图如图 2.

我们的类型设计比文献[7]中的更简单,也更激进.一个激进之处在于将目标机器模型中的数据布局信息应用于类型系统.另一个激进之处在于对数组对象的表示.在文献[7]中,数组对象被刻画成一个不可区分的对象,具有 object 类型.例如

```
int p, q, r, i, j;
struct {int a, *b, *c; } arr[10];
i=1; j=2;
arr[i].b=&p;
arr[i].c=&q;
arr[j].c=&r;
```

$i1: \tau_{12} = \text{simple}((\bot \times \bot)4)$
$i2: \tau_1 = \text{simple}((\tau_7 \times \bot) \times 4)$
$i3: \tau_4 = \text{simple}((\tau_9 \times \bot) \times 4)$
$i4: \tau_5 = \text{simple}((\tau_{10} \times \bot) \times 4)$
$f1: \tau_{13} = \text{simple}((\bot \times \bot) \times 4)$
$f2: \tau_6 = \text{simple}((\tau_{11} \times \bot) \times 4)$
$s1: \tau_7 = \text{struct}([\langle 0,4 \rangle \to \tau_8, \langle 4,4 \rangle \to \tau_9, \langle 8,4 \rangle \to \tau_{10}, \langle 12,4 \rangle \to \tau_{11}])$
$s2: \tau_2 = \text{simple}((\tau_7 \times \bot) \times 4)$
$s3: \tau_7$
$s4: \tau_3 = \text{simple}((\tau_7 \times \bot) \times 4)$
$s1.a: \tau_8 = \text{simple}((\bot \times \bot) \times 4)$
$s3.d: \tau_8$
$s1.b: \tau_9 = \text{simple}((\bot \times \bot) \times 4)$
$s3.e: \tau_9$
$s1.c: \tau_{10} = \text{simple}((\tau_{12} \times \bot) \times 4)$
$s3.f: \tau_{10}$
$s3.g: \tau_{11} = \text{simple}((\tau_{13} \times \bot) \times 4)$

图 4　针对图 1(a)的例子,改进后的分析在目标机器为 IA32 的情况下得到的类型信息

在文献[7]中并不能区分 $arr[i].b$,$arr[i].c$,$arr[j].c$ 为 3 个不同的对象,因此给出的分析结果是整个数组可能指向 p,q,r.我们的类型设计虽然不能区分 $arr[i].c$ 和 $arr[j].c$ 为两个不同的对象,但是可以区分 $arr[i].b$ 和 $arr[j].c$.因此在我们的结果指向图上,整个数组被描述成一个结构体.结构体的域成员 b 指向 p,域成员 c 指向 q 和 r.这种激进的正确性可以由流不敏感分析的不可注销性保证,对某个数组元素的赋值导致整个数组被弱更新[12].如何为数组对象分配类型变量将在第 5 节类型推导中介绍.

4.2 类型规则

类型规则规定了指针分析应该具有的性质,同时也用来保证程序是类型良好的(well-typed).一个程序是类型良好的,是指静态指针分析的结果可以保守地描述程序实际运行时的动态存储构造[3].我们先来定义一些类型变量之间的基本关系,然后利用这些关系去构造类型规则.

解释 1. 类型变量之间的全等价关系=.

$t_1 = t_2$ 表示类型变量 t_1,t_2 对应的存储对象或者函数对象是别名的,也就是说它们对应同一块存储空间.

定义 2. 类型变量之间的部分等价关系$\sqsubseteq$.

$$t_1 \sqsubseteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$
$$(t_1 \times t_2) \sqsubseteq (t_3 \times t_4) \Leftrightarrow (t_1 \sqsubseteq t_3) \wedge (t_2 \sqsubseteq t_4).$$

定义 3. 类型变量之间的部分序关系$\trianglelefteq$.$t_1 \trianglelefteq t_2 \Leftrightarrow$ 类型变量 t_1,t_2 指向对象的类型变量 o_1,o_2 是部

分等价的，即 $o_1 \sqsubseteq o_2$.

$t_1 \trianglelefteq t_2$ 表示类型变量 t_1, t_2 对应的存储对象的指向是条件相同的. 条件相同的概念将在后文解释.

接下来我们定义类型规则. 每条类型规则的含义是，在处理一条程序语句时，我们必须保证相应的约束被满足.

类型规则 1.

$$\frac{A \vdash MEM_1 : \tau_1 \quad A \vdash MEM_2 : \tau_2 \quad \tau_2 \trianglelefteq \tau_1}{A \vdash welltyped(MEM_1 = MEM_2)}$$

A 表示当前的类型环境. 规则的含义是，在经过语句 $MEM_1 = MEM_2$ 之后，为了保证程序是类型良好的，约束 $\tau_2 \trianglelefteq \tau_1$ 必须被满足，即此时 MEM_1 和 MEM_2 的指向条件相同.

部分序关系 $\trianglelefteq$ 的定义要求 τ_1, τ_2 指向对象的类型变量 o_1, o_2 是部分等价的而不是全等价的，是出于对精度的考虑. 例如对程序片段

$a=4;$

$x=a;$

$y=a;$

如果定义 o_1, o_2 具有全等价关系 $o_1 = o_2$，那么 a, x, y 将具有完全相同的指向. 实际上这并不是严格必要的. 如果在整个程序中 a 不指向任何对象，而 x 或 y 在程序的其它部分指向了某个对象 b，那么分析将得到 a 也将指向 b. 而定义 $o_1 \sqsubseteq o_2$ 可以保证分析不会得到 a 指向 b 这个结果[3]中有更详细的解释. 因此 $t_1 \trianglelefteq t_2$ 表示类型变量 t_1, t_2 对应的存储对象的指向是条件相同的，意思是说如果在程序中 t_2 指向了某个对象，那么 t_1 和 t_2 的指向完全相同的；否则 t_2 的指向为空.

类型规则 2.

$$\frac{A \vdash MEM_1 : \mathrm{simple}((\tau \times _) \times size)/\mathrm{object}((\tau \times _) \quad A \vdash MEM_2 : \tau}{A \vdash welltyped(MEM_1 = \& MEM_2)}$$

$$\frac{A \vdash MEM_1 : \mathrm{struct}(m) \quad First(m) = \mathrm{simple}((\tau \times _) \times size)/\mathrm{object}((\tau \times _) \quad A \vdash MEM_2 : \tau}{A \vdash welltyped(MEM_1 = \& MEM_2)}$$

这两条子规则之间是析取的关系，共同组成了规则 2. 第 1 条子规则说明，在经过语句 $MEM_1 = \& MEM_2$ 之后，MEM_1 的指向和 MEM_2 别名. 第 2 条子规则说明，如果 MEM_1 是一个结构体，那么它的起始域成员的指向和 MEM_2 别名. 这样做是合理的. 由于我们的中间表示上的赋值语句都是在内定义数据类型之间发生，因此 MEM_1 具有 struct 类型只能来自于两种情况：

(1) 将结构体的首地址作为其第一个域成员的地址进行解引用. 如图 1(a) 中最后一行，如果对 $i2$ 进行解引用实际上是为了使用 $s3.d$.

(2) 在合并时由标量 simple 提升而来. 提升的方式就是将此标量作为结构体唯一的域成员.

标量的提升将在 5.2 节介绍. 无论哪种情况，取 struct 类型的第 1 个域成员的指向作为 struct 类型的指向都是可靠的.

类型规则 3.

$$\frac{A \vdash MEM : \tau}{A \vdash welltyped(MEM = allocate(c))}$$

规则的约束只要求 MEM 是一个存储对象而不是函数即可，并不做过多的限制. 这是因为我们并不进行显式的堆建模，所有对堆对象的使用都要通过 MEM 指针.

类型规则 4.

$$\frac{A \vdash MEM : \tau \quad A \vdash MEM_i : \tau_i \quad \forall i \in [1..n].\tau_i \trianglelefteq \tau}{A \vdash welltyped(MEM = op(MEM_1 \cdots MEM_n))}$$

规则要求，在经过此语句之后，所有 MEM 的指向必须条件相同.

类型规则 5.

$$\frac{A \vdash MEM : \mathrm{simple}((_ \times \lambda) \times size)/\mathrm{object}((_ \times \lambda)) \quad A \vdash fun : \lambda}{A \vdash welltyped(MEM = fun(f_1 \cdots f_n) \rightarrow r)}$$

$$\frac{A \vdash MEM_1 : \mathrm{struct}(m) \quad First(m) = \mathrm{simple}((_ \times \lambda)) \times size/\mathrm{object}((_ \times \lambda)) \quad A \vdash fun : \lambda}{A \vdash welltyped(MEM = fun(f_1 \cdots f_n) \rightarrow r)}$$

这两条子规则之间是析取的关系，共同组成了规则 4. 规则针对函数指针赋值. 第一条子规则说明，在经过赋值语句之后，MEM 的指向和函数 $func$ 别名. 第二条子规则说明，如果 MEM 是一个结构体，那么它的起始域成员的指向和函数 $func$ 别名.

类型规则 6.

$$\frac{A \vdash MEM : \tau' \quad A \vdash MEM_i : \tau'_i \quad A \vdash f : \mathrm{lam}(\tau_1 \cdots \tau_n)(\tau_{n+1}) \quad \forall i \in [1..n+1].\tau'_i \trianglelefteq \tau_i \quad \tau_{n+1} \trianglelefteq \tau'}{A \vdash welltyped(MEM = f(MEM_1 \cdots MEM_n))}$$

规则针对函数调用语句. 在经过一条函数调用语句

之后,形参和实参的指向必须条件相同,返回值和 *MEM* 指向也必须条件相同.

这 6 条类型规则保证我们的指针分析是保守的. 与文献[7]的区别是,我们的类型规则允许指针具有 struct 类型. 这是由我们的中间表示和改进的类型体系保证,将在下一节介绍.

5　类型推导

指针分析的过程就等于推导一个类型良好程序的类型环境. 本节将要介绍如何进行类型推导,如何能够保证规则约束成立,以及如何能够保证部分序关系$\trianglelefteq$成立.

Steensgaard 风格的指针分析是一种基于等价的(equivalance-based)分析,类型推导阶段运行在并查(union-find)集数据结构之上. 在类型推导阶段,每个类型变量用实际的不相交集合表示. 两个具有等价关系$=$的类型变量被认为是同一个类型变量,处于一个等价类中,使用并查集上的合并操作来将这两个不相交集合合并成一个集合.

我们设计函数 join 来完成两个类型变量的合并. 对于部分等价关系$\sqsubseteq$,函数 cjoin 完成两个类型变量的条件合并. 对于部分序关系$\trianglelefteq$,函数 cjoin_ref 完成两个类型变量的指向的条件合并. 利用函数 cjoin 和 cjoin_ref 可以保证部分序关系$\sqsubseteq$和$\trianglelefteq$成立. 我们先定义推导规则,然后再解释合并函数. 合并函数以及推导规则中所用到的辅助函数的伪代码定义在附录中. 类型推导过程实际上就是指针分析的实施过程,而推导规则就相当于具体的分析算法.

5.1　推导规则

推导规则 1.

$MEM_1 = MEM_2$

let t_1 = GetECR(MEM_1)

t_2 = GetECR(MEM_2) in

if $t_1 \neq t_2$

cjoin_ref (t_1, t_2)

函数 GetECR 获得一个存储对象或函数对象的类型变量,用不相交集合表示.

推导规则 2.

$MEM_1 = \& MEM_2$

let t_1 = GetECR(MEM_1)

t_2 = GetECR(MEM_2) in

if type(t_1) = simple(($\tau_1 \times \lambda_1$) $\times$ *size*) or

object($\tau_1 \times \lambda_1$)

join(τ_1, t_2)

else if type(t_1) = struct(m)

let t_3 = First(type(m)) in

if type(t_3) = simple(($\tau_3 \times \lambda_3$) $\times$ *size*)

join(τ_3, t_2)

else

promote_to_object(t_3)

let object($\tau_3 \times \lambda_3$) = type(t_3) in

join (τ_3, t_2)

First 用来获得映射 m 的第一个域成员. 如果此域成员还是 struct 类型,那么将其提升为 object 类型.

推导规则 3.

$MEM = allocate(c)$

let t_1 = GetECR(MEM) in

if type(t_1) = $\perp$

settype(e_1, blank)

MEM 是一个存储对象而不是函数.

推导规则 4.

$MEM = op(MEM_1 \cdots MEM_n)$

let t_1 = GetECR(MEM) in

for $i = 1$ to n do

let t_2 = GetECR(MEM_i) in

if $t_1 \neq t_2$

cjoin_ref(t_1, t_2)

表达式的每个操作数都具有和 *MEM* 条件相同的指向.

推导规则 5.

$MEM = fun(f_1 \cdots f_n) \rightarrow r$

let t_1 = GetECR(MEM_1)

t_2 = GetECR(MEM_2) in

if type(t_1) = simple(($\tau_1 \times \lambda_1$) $\times$ *size*) or

object($\tau_1 \times \lambda_1$)

then cjoin(λ_1, t_2)

else if type(t_1) = struct(m)

then

let t_3 = First(type(m)) in

if type(t_3) = simple(($\tau_3 \times \lambda_3$) $\times$ *size*)

join(λ_3, t_2)

else

promote_to_object(t_3)

let object($\tau_3 \times \lambda_3$) = type(t_3) in

join (λ_3, t_2)

推导规则 4 与推导规则 2 类似. 函数指针赋值相当于取某个函数的地址,然后赋给函数指针变量.

推导规则 6.

$MEM = f(f_1 \cdots f_n)$

let t = GetECR(MEM)

lam($e_1 \cdots e_n$)(e_{n+1}) = type(GetECR(f)) in

```
for i∈[1…n] do
  cjoin_ref(e_1, t_1)
cjoin_ref (t_1, e_{n+1})
```

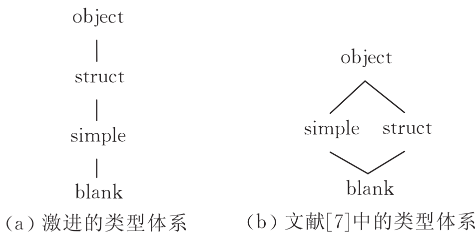
函数调用相当于一些列赋值语句，首先将调用点处实参赋值给相应的形参，然后将函数返回值赋值给 *MEM*.

5.2　join 函数

合并函数 join 首先使用并查集上的合并操作将表示两个类型变量的不相交集合合并成一个集合，然后再调用 unify 函数计算合并后的类型.

5.2.1　类型体系

我们先定义一个类型体系(type hierarchy)来帮助我们计算合并后类型变量的类型，如图 5(a).



(a) 激进的类型体系　(b) 文献[7]中的类型体系

图 5　类型体系

图 5(b)是文献[7]中规定的类型体系. Steensgaard 规定两个等价的类型变量被合并之后的类型在类型体系上要等于或高于这两个变量的类型. 类型为 simple 和 struct 的类型变量被合并会形成一个类型为 object 的类型变量. 例如

```
int  i1, *i2;
struct {int a, *b, *c; } s1;
i2=&i1;
i2=(int*) &s1;
```

根据类型规则 2，我们要保证 *i*2 的指向和 *i*1 以及 *s*1 别名. 这样会导致 *i*1 和 *s*1 的类型变量被合并成 object的类型变量，进而导致 *s*1 的所有域成员都要被合并.

激进的类型体系可以避免这一点，因为我们只将 *i*1 的类型变量提升为仅携带一个 int 型域成员的 struct 类型，成员的存储布局与目标机器模型中的数据布局有关. 然后合并两个 struct 类型的类型变量，结果还是 struct 类型.

5.2.2　惰性域成员映射生成

函数 Get_ECR 用来得到一个存储对象或函数对象的类型变量. 对于域成员，Get_ECR 使用惰性域成员映射生成(Lazy Field Map Generation)技术来获得该域成员的类型变量. 惰性域成员映射生成技术在遇到对域成员的使用时才将此域成员加入至结构体的域成员映射中. 采用惰性域成员映射生成技术能够降低分析时的时空开销. 首先如果一个结构体变量的某些域成员根本没有在程序中使用，我们就不需要为这些域成员建立映射以及分配等价类表示. 其次当两个结构体对象被合并时(union 函数)，没有建立映射的域成员就不需要参与合并，节省时间开销. 在此之后如果出现对这个域成员的使用，直接在新合并的结构体对象中建立映射，可以避免在原来两个结构体对象中分别建立映射.

5.2.3　数组对象的表示

如第 4 节所述，在文献[7]中，数组对象被刻画成一个不可区分的对象，具有 object 类型. 我们对数组元素的表示方法是忽略其下标. 例如

```
int p, q, r, i, j;
struct {int a, *b, *c;}  arr[10];
arr[i].b=&p;
arr[i].c=&q;
arr[j].c=&r;
```

在文献[7]中 *arr*[1].*b*，*arr*[1].*c* 和 *arr*[2].*c* 表示为同一个不可区分的对象，假设记为 *arr*. 我们的表示则可以区分结构体数组元素的不同域成员，记为 *arr*.*b*，*arr*.*c*. 这种激进的正确性可以由流不敏感分析的不可注销性保证，对某个数组元素的赋值导致整个数组被弱更新[11]. 获得数组对象的类型变量也是由函数 Get_ECR 来完成.

5.2.4　成员合并

在合并两个 struct 类型对象时，union 函数用来合并它们可能重叠的域成员，并得到新的域成员映射. 这里我们利用目标机器模型中的数据布局信息可以得到精确的存储布局，尽可能减少不重叠域成员的合并，效果如图 2 和图 1(b)的比较. 文献[7]中的方法完全利用标准数据类型等价性来判断域成员的重叠情况. 这样做的一个弱点是对域成员的重叠情况判断不精确. 另一个弱点是必须一次性为两个 struct 类型对象建立起完全的域成员映射. 这是因为重合情况需要从第一个域成员开始顺序判断，只要遇到两个数据类型不同的域成员，后续的域成员都会被认为是相互重合的. 例如图 1(a)中 *s*1.*c* 和 *s*3.*f* 的数据类型不同导致将 *s*3.*g* 判断为与 *s*1.*c* 和 *s*3.*f* 重合. 如果 *s*1 和 *s*3 还有后续的域成员，那么它们都与 *s*1.*c* 和 *s*3.*f* 重合. 因此文献[7]无法使用惰性域成员映射生成技术.

5.3　cjoin_ref 函数

cjoin_ref 函数用来保证部分序关系$\trianglelefteq$成立，即

条件合并两个类型变量的指向. cjoin_ref 的实现一样要以图 5(a)的类型体系为依据. cjoin_ref 两个 simple 类型变量只需要合并它们的指向即可. cjoin_ref 两个 object 类型变量同样也只需要合并它们的指向. 而当某个类型变量是 struct 类型的时候,情况看上去有点复杂. 正如之前介绍的类型规则 2 和推导规则 2 分析的那样,由于中间表示上的赋值语句都是在内定义数据类型之间发生,所有直接对结构体对象的操作都被转化为对其相应域成员的操作. 因此一个指针具有 struct 类型只能来自于两种情况:

(1) 将结构体的首地址作为其第一个域成员的地址,然后对其解引用.

(2) 在 join 合并时由标量 simple 提升而来. 提升的方式就是将此标量作为结构体唯一的域成员.

无论哪种情况,取 struct 类型的指向实际上就是取其第一个域成员的指向. 如果第一个域成员还是 struct 类型,就将其提升为 object 类型.

在文献[7]中,保证约束 $a \trianglelefteq b$ 成立的一个必要条件是在图 5(b)所示类型体系上,b 的类型要高于或等于 a. 因此如果 a 的类型是 simple,b 的类型是 struct,那么 b 的类型都要被提升为 object,这会导致原来 b 的域成员全部被合并. 除此之外,如果 a 的类型是 struct,b 的类型是 object 还将会导致 a 的每一个域成员的指向与 b 的指向合并.

6　实验结果

我们在编译器 Open64 中实现了两种域敏感的指针分析,即 Steensgaard 的方法(FS)[7] 以及本文的激进方法(AFS). 连带着 Open64 中原有域不敏感指针分析(FI)[3],我们进行了对比实验. 实验用例如表 1 所示.

表 1 中各列的含义是:

【测试用例】:测试程序名.

【行数】:测试程序行数.

【访存数】:测试程序中访存操作个数.

【域访存数】:访问结构体域成员的访存操作数.

【间接域访存数】:间接访问结构体域成员的访存操作数.

所有的实验用例都来自 SPEC2000 和 SPEC2006 测试程序集. 列【域访存数】表示访问结构体域成员的访存操作数,例如 $s.a$, $p{-}{>}a$. 而列【间接域访存数】表示其中通过指针解引用被访问的结构体域成员操作数. 可以看出大部分域成员操作都是通过指针完成的. 从测试集 SPEC2000 和 SPEC2006 中可以看出,在 C 语言中使用结构体对象是很普遍的,对结构体成员的访存操作占全体访存操作的百分比,最少是 1.2%,最多达 29.2%,平均是 13.7%.

表 1　实验用例

测试用例	行数	访存数	域访存数	间接域访存数
164. gzip	3604	6472	265	244
175. vpr	8694	18641	1820	1649
176. gcc	83475	231765	41166	39809
177. mesa	28863	91881	13471	12755
179. art	809	1900	124	124
181. mcf	900	1996	567	562
183. equake	953	2925	120	20
188. ammp	9681	24422	4272	4272
197. parser	8915	15357	2407	2375
253. perlpmk	7409	104074	18673	17940
254. gap	36631	101273	11485	11480
255. vortex	28634	69600	9423	8062
256. bzip2	25519	4007	87	87
300. twolf	2427	38249	6457	6457
400. perlbench	14617	197962	37804	36912
401. bzip2	67022	12926	2683	2666
403. gcc	8644	522023	90900	89154
429. mcf	193032	2029	594	588
433. milc	931	16118	1613	1432
445. gobmk	7575	102208	5511	5216
456. hmmer	333448	47184	5797	5370
458. sjeng	16506	15836	1473	1411
462. libquantum	7072	4509	748	543
464. h264ref	1776	88695	12878	11583
470. lbm	28887	2518	30	26
482. sphinx3	456	23212	4068	4037

表 2 展示了 3 种指针分析的实验结果. 表 2 中各列的含义是:

【域不敏感】:域不敏感基于合并的指针分析[3].

【域敏感】:域敏感基于合并的指针分析[3].

【激进域敏感】:本文的激进域敏感基于合并的指针分析.

【测试用例】:测试程序名.

【类数】:所有间接访问结构体域成员的访存操作所涉及的别名类个数. 在一个别名类中的访存操作是别名的,它们存取相同的存储位置.

【最大】:最大的别名类中间接域成员访存操作数.

【平均】:平均每个别名类中域成员访存操作数.

【时间】:指针分析过程所用时间.

列【类数】将表 1 中列【间接域访存数】列统计的访存操作分为若干别名类,每个别名类中的语句是有别名的. 这就是指针分析得到的直接结果. 我们之

所以选择列【间接域访存数】作为衡量对象而不选择列【域访存数】是因为区分直接域成员访存操作是否别名是很容易的，而且大部分域成员操作都是通过指针完成的. 列【类数】的数值越大表明两条访存操作的别名几率越小.

列【平均】是用列【类数】去除列【间接域访存数】得来的，代表平均每个别名类中域成员访存操作数. 我们以列【平均】的大小来衡量指针分析的精度. 列【平均】的数值越大说明分析得出两个访存操作别名的可能性越大. 我们根据列【平均】的数值制作了一个归一化的直方图，以激进域敏感分析的【平均】列数值为参考值 1.0，如图 6. 可以看出，域敏感的分析的精度远远高于域不敏感的分析，即它能够区分的是否别名的访存操作要明显多于域不敏感的分析. 激进域敏感的分析在精度上相比于域敏感的分析有不同程度的提高，其中超过半数的例子有大幅度提高，提高精度一倍以上.

表 2　3 种基于合并的指针分析比较

测试用例	域不敏感				域敏感				激进域敏感			
	类数	最大	平均	时间	类数	最大	平均	时间	类数	最大	平均	时间
164.gzip	4	222	61.00	0.03	37	72	6.59	0.04	43	72	5.67	0.04
175.vpr	23	1408	71.69	0.11	222	183	7.42	0.12	258	183	6.39	0.12
176.gcc	89	38388	447.29	1.22	278	37516	143.19	1.65	1502	31759	26.50	1.64
177.mesa	18	12593	708.61	0.46	87	12219	146.60	2.83	1162	471	10.97	2.11
179.art	2	97	62.00	0.01	10	26	12.40	0.01	10	26	12.40	0.01
181.mcf	7	527	80.28	0.01	57	40	9.85	0.02	76	33	7.39	0.02
183.equake	2	12	10.00	0.01	3	12	6.67	0.02	8	5	2.50	0.02
188.ammp	2	4262	2136.00	0.10	91	3330	46.94	0.16	231	1048	18.40	0.13
197.parser	18	2115	131.94	0.08	69	2003	34.42	0.10	225	1489	10.55	0.10
253.perlpmk	14	17821	1281.42	0.47	172	17212	104.30	0.57	971	13971	18.47	0.59
254.gap	9	11459	1275.55	0.50	560	9603	20.50	0.56	1723	6451	6.66	0.56
255.vortex	90	7631	89.57	0.45	326	7506	24.73	0.56	432	4759	18.65	0.53
256.bzip2	2	69	43.50	0.02	2	69	43.50	0.02	7	26	12.42	0.02
300.twolf	1	6457	6457.00	0.19	80	4495	80.71	0.20	162	4065	39.85	0.20
400.perlbench	25	35366	1476.48	0.92	297	33947	124.28	1.15	1697	25410	21.75	1.12
401.bzip2	2	2665	1333.00	0.06	8	2505	333.25	0.09	18	1857	148.11	0.09
403.gcc	308	86495	289.46	2.89	1267	82626	70.36	4.41	4799	60981	18.57	4.19
429.mcf	7	533	84.00	0.01	59	41	9.96	0.02	78	34	7.53	0.01
433.milc	32	1320	44.75	0.09	85	1096	16.84	0.12	141	539	10.15	0.12
445.gobmk	58	4014	89.93	0.78	208	1642	25.07	0.98	499	348	10.45	1.10
456.hmmer	103	4336	55.63	0.26	659	791	8.69	0.34	812	454	7.05	0.31
458.sjeng	27	740	52.25	0.08	160	302	8.81	0.10	197	301	7.16	0.11
462.libquantum	51	393	10.64	0.03	107	139	5.07	0.03	113	139	4.80	0.03
464.h264ref	6	11434	1930.50	0.45	306	9452	37.85	0.66	722	2816	16.04	0.62
470.lbm	1	26	26.00	0.01	5	10	5.2	0.01	5	10	5.20	0.01
482.sphinx3	60	3856	67.28	0.14	265	3005	15.23	0.19	555	849	7.27	0.18

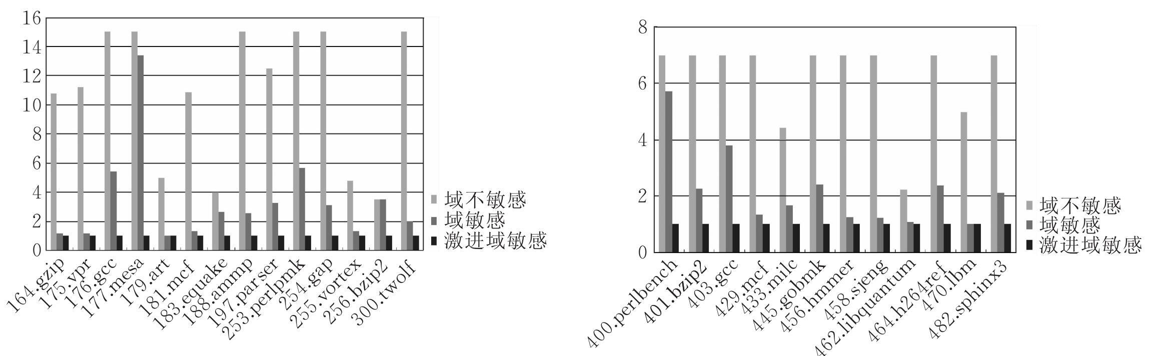


图 6　根据表 2 列【平均】得到的归一化的直方图（以激进域敏感分析的【平均】列数值为参考值 1）

实验表明，在我们选用的 27 个 C 程序中，域不敏感的指针分析所用时间开销要小于域敏感的分析. 两种域敏感分析的时间开销基本相当. AFS 比 FS 做了更激进的分析而时间开销并没有增加的一个原因是 AFS 能够尽量避免域成员之间的合并操作，减少合并开销. 另一个原因是 AFS 分析采用了

惰性域成员映射技术来降低分析的时空开销.

我们还将这3种指针分析结果应用于构建完整的过程调用图.在完整的过程调用图上,对函数指针的调用点被实例化成对该函数指针的所有指向对象的调用,因此会在过程调用图上添加相应的调用边.函数指针调用点,对于测试用例177.mesa 3种指针分析效果相差极为明显.177.mesa中共有671处函数指针调用,利用FI分析结果会向过程调用图上增加297253条调用边,平均每处函数指针调用增加了443条边.利用FS分析结果会向过程调用图上增加295240条调用边,平均每处函数指针调用增加了440条边,效果相比FI几乎没有提高.而利用我们的AFS分析结果会向过程调用图上增加1537条调用边,平均每处函数指针调用只增加了2.29条边,精度大约提高200倍,效果提高显著.分析177.mesa程序特征我们发现整个程序都是在围绕着结构体指针

```
GLcontext  *CC
```

对结构体进行访存操作.绝大多数函数指针都是*CC*指向的结构体对象的域成员变量.由于FI分析是域不敏感的,将*CC*指向的结构体对象的所有域成员看成一个对象,因此每个域成员函数指针都会指向所有域成员指向的函数.FS分析虽然是域敏感的,但是对177.mesa这个例子效果并不显著.进一步分析原因得出,*CC*指向的对象是通过调用函数gl_get_thread_context获得的.该函数的定义如下.

```
GLcontext *gl_get_thread_context(void)
{
   pid_t id;
   int i;
   id=getpid();
   for (i=0;i<MAX_THREADS;i++) {
      if (ThreadID[i]==id) {
         return ThreadContext[i];
      }
   }
   return NULL;
}
```

可以看出,该函数返回的是一个数组元素.由于FS分析将数组刻画成一个不可区分的对象,如4.1节所述,因此*CC*指向的是一个object对象.也就是说FS分析无法区分CC指向的结构体成员.AFS分析则可以做到这一点.

7 相关工作

基于合并方式(unification-based)的指针分析[3]或者称为Steensgaad风格的指针分析应用非常广泛.Steensgaard风格的指针分析是一种流不敏感的指针分析,优点是具有接近线性的时间复杂度.Das对文献[3]进行了改进[8,12],允许相互赋值的两个指针的第一级指向不被合并,而从第二级指向开始合并,因此分析的精度要高于[3],但是时空效率也由此降低.Fahndrich[9]提出了一种上下文敏感的Steensgaad风格的指针分析.

Steensgaard本人对其工作[3]进行了改进,形成域敏感版本[7].文献[7]中采用了最大公共子前缀的技术来建立结构体类型之间的对应关系.Yong[10]将Steensgaard的思想进一步扩展,给出了3种不同精度的方式解析结构体类型之间的对应关系,并将这些方式推广到各种指针分析,其中最精确的方式与文献[7]相当,都是最大公共子前缀思想.Yong[10]还给出了一种利用目标机器信息进行域敏感分析的模式,但是他认为这样做的一个缺陷是指针分析工具将不可移植,而且也没有讨论如何利用目标机器信息改进基于合并方式的指针分析.这种改进并不是一种平凡的改进.Pearce[5]用最大公共前缀的思想改进了Andersen风格的方法,并且使用了与文献[10]不同的表示方式来表示结构体对象的域成员,因此算法的效率更高.

8 结 论

本文提出一种激进的基于合并的域敏感指针分析.与以往基于合并的域敏感指针分析[7]相比,本方法在如下方面做了改进:

(1)利用目标机器模型中的数据布局信息进行高层分析,精确刻画结构体域成员的存储布局.受益于编译器Open64灵活的框架,激进的同时又具有可移植性.

(2)改进的类型体系尽量避免在合并类型变量时造成精度损失.

(3)采用惰性域成员映射生成技术尽可能地推迟建立域成员映射,节省分析的时空开销.

(4)对结构体数组对象的激进表示.

大量实验数据表明,本方法分析精度显著高于Steensgaard的方法[7],而运行开销几乎相当,互有高低.

参 考 文 献

[1] Ghiya R, Lavery D, Sehr D. On the importance of points-to

analysis and other memory disambiguation methods for C programs//Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. Snowbird, Utah, United States, 2001: 47-58

[2] Andersen L O. Program analysis and specialization for the C programming language[Ph. D. dissertation]. University of Copenhagen, DIKU, 1994

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

[4] Hardekopf B, Lin C. 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. San Diego, California, USA, 2007: 290-299

[5] Pearce D J, Kelly P H J, Hankin C. Efficient field-sensitive pointer analysis of C. ACM Transactions on Programming Languages and Systems (TOPLAS), 2007, 30(1): 4

[6] Pereira F M Q, Berlin D. Wave propagation and deep propagation for pointer analysis//Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Seattle, WA, USA, 2009: 126-135

[7] Steensgaard B. Points-to analysis by type inference of programs with structures and unions//Proceedings of the 6th International Conference on Compiler Construction. London. UK: Springer-Verlag, 1996: 136-150

[8] Das M. Unification-based pointer analysis with directional assignments//Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. Vancouver, British Columbia, Canada, 2000: 35-46

[9] Fahndrich M, Rehof J, Das M. Scalable context-sensitive flow analysis using instantiation constraints//Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. Vancouver, British Columbia, Canada, 2000: 253-263

[10] Yong S H, Horwitz S, Reps T. Pointer analysis for programs with structures and casting//Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. Atlanta, Georgia, United States, 1999: 91-103

[11] Chase D R, Wegman M, Zadeck F K. Analysis of pointers and structures//Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. White Plains, New York, United States, 1990: 296-310

[12] Das M, Liblit B, Fahndrich M, et al. Estimating the impact of scalable pointer analysis on optimization//Proceedings of the 8th International Symposium on Static Analsis, 2001. London. UK: Springer-Verlag, 2001: 260-278

附 录.

合并函数以及推导规则中用到的辅助函数的伪代码.

```
cjoin(e1, e2)
  if type(e2) = ⊥
    pending(e2) ← {e1} ∪ pending(e2)
  else
    join(e1, e2)

join(e1, e2)
  if e1 ≠ e2
    let t1 ← type(e1)
      t2 ← type(e2) in
      e ← ecr-union(e1, e2)
      if t1 = ⊥
        type(e) ← t2
        if t2 = ⊥
          pending(e) ← pending (e1) ∪ pending(e2)
        else
          for x ∈ pending (e1) do
            join(e, x)
      else
        type(e) ← t1
        if t2 = ⊥
          for x ∈ pending(e2) do
            join(e, x)
        else
          unify(e1, e2)

cjoin_ref (e1, e2)
  if type(e1) = blank
    settype(e1, type(e2))
  else if type(e2) = blank
    settype(e2, type(e1))
  else case type(e1) of
    [simple((τ1 × λ1) × size1)]:
      case type(e2) of
        [simple ((τ2 × λ2) × size2))]:
          cjoin(τ1, τ2)
          cjoin(λ1, λ2)
        [struct(m2)]:
          let ⟨o2, s2⟩ → τ2 = First(m2) in
            if type(τ2) = struct(m)
              promote_to_object(τ2)
            if type(τ2) = simple ((τ × λ) × size)) or
                          object (τ × λ)
              cjoin(τ1, τ)
              cjoin(λ1, λ)
          [object(τ2 × λ2)]:
            cjoin(τ1, τ2)
            cjoin(λ1, λ2)
    [struct(m1)]:
      case type(e2) of
        [simple ((τ2 × λ2) × size2))]:
          let ⟨o1, s1⟩ → τ1 = First(m1) in
            if type(τ1) = struct(m)
              promote_to_object(τ1)
            if type(τ1) = simple ((τ × λ) × size)) or
                          object (τ × λ)
```

```
              cjoin(τ, τ2)
              cjoin(λ, λ2)
        [struct(m2)]:
          let ⟨o1,s1⟩→τ1 = First(m1)
              ⟨o2,s2⟩→τ2 = First(m2) in
            if type(τ1) = struct(m1)
              promote_to_object(τ1)
            if type(τ2) = struct(m2)
              promote_to_object(τ2)
            if type(τ1) = simple ((τ3×λ3)×size3)) or
                          object (τ3×λ3) and
              type(τ2) = simple ((τ4×λ4)×size)) or
                          object (τ4×λ4)
              cjoin(τ3, τ4)
              cjoin(λ3, λ4)
        [object(τ2×λ2)]:
          let ⟨o1,s1⟩→τ1 = First(m1) in
            if type(τ1) = struct(m)
              promote_to_object(τ1)
            if type(τ1) = simple ((τ×λ)×size)) or
                          object (τ×λ)
              cjoin(τ, τ2)
              cjoin(λ, λ2)
      [object(τ1×λ1)]:
        case type(e2) of
          [simple ((τ2×λ2)×size2))]:
            cjoin(τ1, τ2)
            cjoin(λ1, λ2)
          [struct(m2)]:
            let ⟨o2,s2⟩→τ2 = First(m2) in
              if type(τ2) = struct(m)
                promote_to_object(τ2)
              if type(τ2) = simple ((τ×λ)×size)) or
                            object (τ×λ)
                cjoin(τ1, τ)
                cjoin(λ1, λ)
            [object(τ2×λ2)]:
              cjoin(τ1, τ2)
              cjoin(λ1, λ2)

unify (e1,e2)
  case type(e1) of
    [simple((τ1×λ1)×size1)]:
      case type(e2) of
        [simple ((τ2×λ2)×size2))]:
          join(τ1, τ2)
          join(λ1, λ2)
          if (size1 < size2)
          then size1 ← size2
          else size2 ← size1
        [struct(m2)]:
          promote_to_struct(e1)
          let struct(m1) = type(e1) in
             union (m1,m2)
        [object(τ2×λ2)]:
          join(τ1, τ2)
          join(λ1, λ2)
          promote_to_object(e1)
    [struct(m1)]:
      case type(e2) of
        [simple ((τ2×λ2)×size2))]:
          promote_to_struct(e2)
          let struct(m2) = type(e2) in
            union (m1,m2)
        [struct(m2)]:
          union (m1,m2)
        [object(τ2×λ2)]:
          promote_to_object(e1)
          let object(τ1×λ1) = type(e1) in
             join(τ1, τ2)
             join(λ1, λ2)
    [object(τ1×λ1)]:
      case type(e2) of
        [simple ((τ2×λ2)×size2))]:
          join(τ1, τ2)
          join(λ1, λ2)
          promote_to_object(e2)
        [struct(m2)]:
          promote_to_object(e2)
          let object(τ2×λ2) = type(e2) in
             join(τ1, τ2)
             join(λ1, λ2)
        [object(τ2×λ2)]:
          join(τ1, τ2)
          join(λ1, λ2)
    [lam(e1…en)(en+1)]:
      let lam(e'1…e'n)(e'n+1) = type(e2) in
        for i ∈ [1…n] do
          cjoin_ref(ei, e'i)

union (m1,m2, t)
  let map1 = type(m1)
      map2 = type(m2)
      map = NewMAP() in
    while map1 not empty or
          map2 not empty
      let ⟨o1,s1⟩→τ1 = First(map1)
          ⟨o2,s2⟩→τ2 = First(map2)
          ⟨o,s⟩→τ = Last(map) in
          if overlap(⟨o1,s1⟩,⟨o2,s2⟩)
            let ⟨o3,s3⟩ = merge(⟨o1,s1⟩,⟨o2,s2⟩)
                τ3 = MakeECR(1) in
              join (τ3, τ1)
              join (τ3, τ2)
              remove(map1, ⟨o1,s1⟩→τ1)
              remove(map2, ⟨o2,s2⟩→τ2)
              if not overlap(⟨o3,s3⟩,⟨o,s⟩)
                insert(map, ⟨o3,s3⟩→τ3)
              else
                let ⟨o4,s4⟩ = merge(⟨o3,s3⟩,⟨o,s⟩)
                    τ4 = MakeECR(1) in
                    join (τ4, τ3)
                    join (τ4, τ)
                    remove(map, ⟨o,s⟩→τ)
                    insert(map, ⟨o4,s4⟩→τ4)
          else
            if in_front_of(⟨o1,s1⟩,⟨o2,s2⟩)
              remove(map1, ⟨o1,s1⟩→τ1)
```

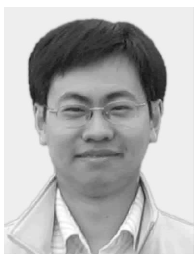
```
        if not overlap(⟨o1,s1⟩,⟨o,s⟩)
          insert(map, ⟨o1,s1⟩→τ1)
        else
          let ⟨o3,s3⟩=merge(⟨o1,s1⟩,⟨o,s⟩)
                τ3=MakeECR(1) in
                join (τ3, τ1)
                join (τ3, τ)
                remove(map, ⟨o,s⟩→τ)
                insert(map, ⟨o3,s3⟩→τ3)
      else
        remove(map2, ⟨o2,s2⟩→τ2)
        if not overlap(⟨o2,s2⟩,⟨o,s⟩)
          insert(map, ⟨o2,s2⟩→τ2)
        else
          let ⟨o3,s3⟩=merge(⟨o2,s2⟩,⟨o,s⟩)
            τ3=MakeECR(1) in
            join (τ3, τ2)
            join (τ3, τ)
            remove(map, ⟨o,s⟩→τ)
            insert(map, ⟨o3,s3⟩→τ3)
    type(m1)←map
    type(m2)←map

GetECR (MEM)
  case form(MEM) of
    [x]:
      return ecr(x)
    [*MEM1]:
      case type(GetECR(MEM1)) of
        [simple ((τ1×λ1)×size1))]:
          return τ1
        [struct(m1)]:
          let ⟨0,sizeof(*MEM1)⟩→e=First(type(m)) in
            if type(e)=simple ((τ×λ)×size)) or
                       object (τ×λ)
              return τ
            else if type(e)=struct(m)
              promote_to_object(e)
              let object (τ×λ)=type(e) in
                return τ
        [object(τ1×λ1)]:
          return τ1
    [MEM1[MEM2]]:
      case type(GetECR(MEM1)) of
        [simple ((τ1×λ1)×size1))]:
          return τ1
        [struct(m1)]:
          let ⟨0,sizeof(*MEM1)⟩→e=First(type(m)) in
            if type(e)=simple ((τ×λ)×size)) or
                       object (τ×λ)
              return τ
            else if type(e)=struct(m)
              promote_to_object(e)
              let object (τ×λ)=type(e) in
                return τ
        [object(τ1×λ1)]:
          return τ1

    [MEM1.⟨offset,size⟩]:
      let e=GetECR(MEM1) in
        case type(e) of
          [simple ((τ1×λ1)×size1))]:
            promote_to_struct(e)
            let struct(m1)=type(e) in
              let τ=Find_or_enter(type (m1), ⟨o,s⟩) in
                return τ
          [struct(m1)]:
            let τ=Find_or_enter(type (m1), ⟨o,s⟩) in
              return τ
          [object(τ1×λ1)]:
            return τ1

promote_to_struct (e)
  if type(e)=simple (τ×λ×size)
    let map=NewMAP()
      t=MakeECR(1) in
        insert(map, ⟨0,size⟩→e)
        cjoin_ref(t, e)
        settype(e, struct(map))

promote_to_object (e)
  case type(e) of
    [simple ((τ×λ)×size))]:
      type(e)←object(τ×λ)
    [struct(m)]:
      let τ=⊥ and λ=⊥ in
        type(e)←object(τ×λ)
        for each ⟨o2,s2⟩→τ2 of m do
          join(e, τ2)
```



YU Hong-Tao, born in 1981, Ph. D. candidate. His research interests include program analysis and compiling optimization.

ZHANG Zhao-Qing, born in 1938, professor, Ph. D. supervisor. Her research interests include advanced compiling technology and related tools.

Backgroud

Pointer analysis is a kind of program analysis technique that determines which memory locations a pointer may point to during the execution of a given program. Field-sensitivity is used to describe whether a pointer analysis needs to distinguish different field members. Field-insensitive pointer analysis considers all fields of one structural object as the same object. On the contrary, field-sensitive pointer analysis considers different fields as different objects. Based on the authors' investigation to famous benchmarks, field-sensitivity is significant to pointer analysis for C, especially for C++. Most of existed work of improving field-sensitivity does not make use of target machine architecture thus they must loss some precision. Only a few work takes target machine architecture into consider, however, their work cannot be applied directly to the unification-based pointer analysis which is widely used in product compliers .

This paper proposes an aggressive field-sensitive unification-based pointer analysis. Different from existed methods, the method takes target machine architecture into consider in the phase of high-level analysis in order to precisely distinguish fields of structure objects. In the method, a field of a structural object is aggressively represented by a pair of offset from its base structure and size of its own data type. This paper also improves the original type system to obtain further precision. All structural memory operations are flattened to a series of scalar memory operations based on the target machine information to guarantee the correctness of type inference system. Lots of experiments indicate that the proposed method is more precise than the existed method while maintaining almost the same efficiency. Furthermore, the method is portable since the authors have implemented the aggressive field representation on the intermediate representation of the compiler.

This research is attached to the project "The Compiler-Based Framework for the Development of High-Reliable Software". The whole project is supported by a grant from the National High Technology Research and Development Program of China (863 Program) (No. 2008AA01Z115). The project aims at improving software reliability, including the detection of software defects, program verification and debugging based on compiling technology. The work of this paper is to improve the precision of the pointer analysis of our baseline compiler Open64 in order to construct precise SSA form for the consequently analysis, e. g. program slicing, uninitialized reference checking and so on.