文章编号:1671-4229(2019)02-0001-09

分离逻辑的技术基础与研究现状

王捍贫[1,2]，张博闻[2]

(1. 广州大学 计算机科学与网络工程学院，广东 广州　510006；
2. 北京大学 a. 信息科学技术学院软件研究所；b. 高可信软件技术教育部重点实验室，北京　100871)

摘　要：随着计算机系统复杂性的日渐增加，可靠性正成为计算机软件理论中新的研究热点. 运用数理逻辑中的推理方法，研究人员可以严格分析和验证计算机程序的正确性. 分离逻辑作为一种分析共享可操作存储程序的新方法，近年来逐渐发展成为针对可扩展程序的主流验证技术之一. 文章阐述分离逻辑的研究背景、理论基础以及验证原理，详细介绍其对存储空间抽象描述的思想，通过实例说明分离逻辑的推导策略. 特别地，介绍了双向诱导推理对循环链表程序片段的验证过程，展现了在未知系统当前状态的情况下，分离逻辑验证未完成代码，甚至代码片段正确性的可行性. 文章还综述了诸多基于分离逻辑的各种拓展，以增加对分离逻辑研究状况的了解.

关键词：分离逻辑；形式化方法；双向诱导推理；程序验证

中图分类号：TP 311　　**文献标志码**：A

1　研究背景

1.1　形式化验证方法简介

作为计算机系统的灵魂，软件中存在的任何缺陷都可能成为隐患，以至于对整个计算机系统造成致命影响. 目前，软件质量的可靠性保障方法正在受到工业界和学术界的广泛关注. 经过多年努力，相继出现了很多相关的工具和技术，它们有的应用于软件开发前的程序架构，有的用于开发期间的实现与测试，还有的用于对已发布的软件进行分析和验证等. 但是对于强安全需求的(safty critical)计算机系统，例如核电站的温控系统、导弹的制导系统等等，不允许程序中隐藏任何潜在错误，否则将会造成巨大的财产损失，甚至对社会造成不可预计的灾难.

形式化验证方法通过运用严格的数学方法分析和验证计算机程序的正确性[1]，从数学角度保证程序中不存在错误，通常使用的方法有公理推导、模型检验(model checking)、约束求解、抽象解释和符号执行等. 形式化验证的优点是，其描述计算机软、硬件及其性质的语言是无歧义的，构造和验证计算机软、硬件的方法是严格的，因此，该技术被广泛应用于验证强安全需求的计算机系统的正确性.

1.2　分离逻辑的研究背景

对程序进行形式化验证的过程中，逻辑断言是用来描述程序状态的主要技术. 图灵(Turing)[2]曾在 1949 年尝试运用断言来验证特定程序的正确性，随后 Floyd[3] 和 Hoare[4] 在 20 世纪 60 年代将该方法推广到程序证明的一般形式，建立了新的逻辑系统，称为 Hoare 逻辑系统(也称 Floyd-Hoare 逻辑). Hoare 逻辑的中心特征是 Hoare 三元组，为命令式程序的规范说明，定义了程序执行前后系统状态应满足的属性. Hoare 逻辑系统对于处理简单原始的数据类型非常有效(例如：整数或字符串)，但 Hoare 不便于描述程序中的地址操作，因此，在处理包含指针的结构化数据时，指针操作虽简单直观，但验证过程会相当复杂晦涩[5].

分离逻辑(Separation Logic)是一种推导共享可操作存储程序的新方法，最初由 Reynolds[6] 和 Ishtiaq 等[7]提出，该逻辑系统是 Hoare 逻辑的一个扩展，其中心特征也是三元组. 其基本形式为

$$\{P\}\,C\,\{Q\},$$

收稿日期：2019－04－13；　**修回日期**：2019－04－20

基金项目：国家自然科学基金资助项目(61170299，61370053 和 61572003)

作者简介：王捍贫(1964—)，男，教授，博士生导师，博士. E-mail：whpxhy@pku.edu.cn

其中 P 称为前置断言,Q 为后置断言,C 是指令. 直观描述为:若在 C 执行前,P 成立. 则当执行终止后(若终止),Q 成立. 例如,下述规范说明

$$\{x==N\}\ code\ \{x==N\wedge y==N!\}$$

可以应用到"将变量 x 值的阶乘赋值于 y" 程序的验证中. 分离逻辑通过解决潜在的指针别名问题,从而简化对包含共享可操作数据结构程序的验证.

在过去的15年中,分离逻辑发展成为针对可扩展程序(scalable program)的主流验证技术[8]. 在LICS、POPL、JACM 等顶尖国际会议和期刊上,几乎年年都有相关文章发表. 分离逻辑之所以能够成功,是因为它能够将软件工程师对于计算机共享可操作存储程序的概念模型与逻辑中解释语句正确性的逻辑模型相结合,从而形成一整套证明系统来帮助解决可操作存储程序推理任务的关键问题.

分离逻辑可以对含有各类链表、树以及图等复杂数据结构的程序进行验证,如链表的排序、完全二叉树的操作、堆排序、AVL 树等等. 还有很多用于实际程序的验证,比如验证 Schorr-Waite 算法与 DFS 算法的等价性[9],Schorr-Waite 是一种基于图的垃圾回收的算法,一般的垃圾回收算法使用深度优先遍历(DFS)搜索,需使用栈记录当前访问路径. 而 Schorr-Waite 程序不使用栈,仅使用图本身的指针实现回溯,这种数据结构可以被分离逻辑简单直接的表示. 此外,还有验证内存管理系统中 Cheney 的复制垃圾回收器(Copying Garbage Collector)程序的正确性[10];运用 SpaceInvader 分析器验证 Linux 和 Windows 系统中近1万个指针的安全性[11];以及在该验证器的基础上由微软开发的 SLAyer,甚至可以在 Windows 的设备驱动程序中找出10多个隐藏的内存安全错误等等[12]. 这些分离逻辑在学术和商业上的应用,很好地说明了其具有很强的实用性.

由于篇幅有限,本文将简要阐述分离逻辑的基本方法,包括该逻辑系统对存储的抽象描述方法、推理过程,以及在自动化验证技术上的革新,并会简要概括分离逻辑的一些拓展及相关工作.

2 理论基础及验证实例

2.1 分离逻辑对存储空间的抽象描述

分离逻辑对于 Hoare 逻辑的扩展在于,能够直观地分析程序中复杂的动态内存变化,这种分析能力依赖于该逻辑系统对计算机程序访存过程的描述. 分离逻辑引入栈(Stack)和堆(Heap)的概念,与数据结构中提到的堆和栈不同,分离逻辑中栈表示"变量→地址"和"变量→值"的映射,堆则表示"地址→值"的映射. 运用堆和栈,分离逻辑系统不仅可以模拟程序直接访问寄存器中的值,还可以模拟"变量→地址→值"的一般情况[13]. 可以用下面的例子:

$$\{x\mapsto 0 * y\mapsto 0\}$$
$$[x]=y;$$
$$[y]=x$$
$$\{x\mapsto y * y\mapsto x\}$$

作为规范说明来描述两个内存地址连接成循环链表. 其中 $x\mapsto v$ 表示指针变量 x 保存着指向"内存中已分配空间" v 所对应的地址;指令 $[x]=y$ 表示更改 x 引用地址中的内容,使其值更新为 v';通过使用"$*$"而不是通常的布尔连接符"$\wedge$",确保 x 和 y 不出现别名混淆,即多个变量指向同一地址,由此在后置条件下就可以描述一个双元素的循环列表[14]. 上述解释可参考图1理解.

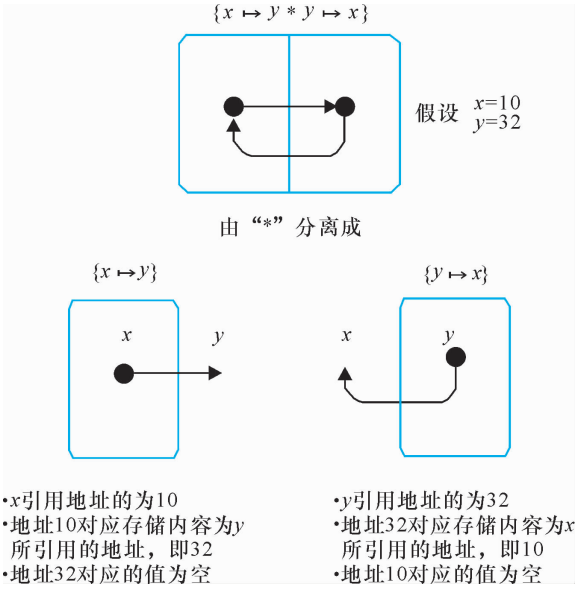

图1 循环链表语义示意图

Fig. 1 The semantics of cyclic linked list

根据分离逻辑对存储空间的描述,下面阐述该逻辑系统语义域,以此来划分它在语义上的讨论范围. 一般的程序语义验证系统不考虑浮点型,因此,本文所描述的分离逻辑系统中的数值只考虑整型,不考虑浮点型,并且该逻辑系统的运算不

包含除法. 程序中变量的值可能为整数或地址:

$$Value = Integers \cup Addresses.$$

地址在计算机中以整型的数据类型存储:

$$Addresses \subseteq Integers;$$

堆表示“地址与值”的映射关系:

$$Heap = \cup_{A \to Addresses}(A \to Values),A \text{ 是有限集;}$$

栈表示“变量→值”的映射关系(包括“变量→地址”):

$$Stack_V = V \to Values,V \text{ 是代表 } Variables \text{ 的有限集;}$$

状态($States_V$)表示当前系统中的存储状态,以栈和堆的笛卡尔积表示:

$$States_V = Stacks_V \times Heaps.$$

为方便讨论,在本文中规定以 s 表示栈,以 h 表示堆. (s,h)表示当前系统的存储状态.

2.2　分离逻辑的语法及断言语言

了解分离逻辑的对计算机存储的抽象,便可以如下定义分离逻辑的语法的 BNF 表示:

$v \in V, n \in Values$

Integer Expns	$i ::=$	$v \mid n \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 \times i_2$
Pointer Expns	$k ::=$	$v \mid v.n \mid \textbf{\textit{null}}$
Boolean Expns	$b ::=$	$T \mid F \mid i_1 = i_2 \mid k_1 = k_2 \mid i_1 \leqslant i_2 \mid b_1 \vee b_2 \mid b \mid b_1 \wedge b_2 \mid b_1 \to b_2$
Expressions	$e ::=$	$i \mid b \mid k$
Assertions	$p,q ::=$	$b \mid k \mapsto e \mid \textbf{\textit{emp}} \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p * q \mid p -\!\!* q \mid \forall v.p \mid \exists v.p$

分离逻辑对于 Hoare 逻辑的拓展主要是通过引入 4 种新的断言形式,以细化对计算机系统存储状态的描述. 下面主要介绍这些新引入的断言.

Assertions　$p,q ::= \cdots$

$\mid \textbf{\textit{emp}}$	空堆
$\mid k \mapsto e$	单堆
$\mid p * q$	分离合取
$\mid p -\!\!* q$	分离蕴含

定义新形式断言的语义要依靠其在状态(s,h)中的赋值,因此,下面结合状态(s,h)来说明新断言的语义. 用$[\![e]\!]_{exp}\ s$ 表示表达式 e 在 s 中的取值,$[\![p]\!]_{asrt} s\ h$ 表示断言 p 在状态(s,h)中可以被满足,由此可定义新断言的语义如下:

空堆

$$[\![emp]\!]_{asrt}\ s\ h \text{ iff } dom\ h = \{\ \}.$$

空堆即堆 h 为空,其中 $dom\ h$ 表示堆 h 的定义域.

单堆

$$[\![e \mapsto e']\!]_{asrt}\ s\ h \text{ iff},$$
$$dom\ h = \{[\![e]\!]_{exp}\ s\} \text{ and } h([\![e]\!]_{exp}\ s) = [\![e']\!]_{exp} s.$$

单堆的意思是系统状态的堆中仅包含了一个单元,且该单元的地址是$[\![e]\!]_{exp}\ s$,内容是 e'. 直观来讲,单堆准确地描述了计算机的存储状态,如图 1 中 $x \mapsto y$ 所代表的“x 指向 y,且 y 指向空”消除了对内存表示的歧义:指针 x 和 y 分别表示某个值$(x=10,y=32)$,x 的值是该变量被分配的地址空间($address$:10),该地址用于存储值 y,但 y 的值并没有被分配空间存储.

分离合取

$[\![p_0 * p_1]\!]_{asrt}\ s\ h$ *iff* $\exists h_0, h_1 \cdot h_0 \perp h_1$ and $h_0 \cdot h_1 = h$ and $[\![p_0]\!]_{asrt}\ s\ h_0$ and $[\![p_1]\!]_{asrt}\ s\ h_1$.

分离合取断言了堆 h 可以被分割成两个互不相交的子堆 h_0,h_1,并且两个子堆分别满足断言 p_0 和 p_1,其中 $h_0 \perp h_1$ 代表两个堆互不相交. 分离合取是分离逻辑很重要的一个断言形式,该断言可以分离堆和内存,但它并不会分割变量与值的关联. 如图 1 中 $x \mapsto y * y \mapsto x$ 就断言了两个不相交的堆的合取. 由上述单堆的定义可知,不考虑分离合取的连接词,就有“x 指向 y,且 y 指向空”以及“y 指向 x,且 x 指向空”这两个形式的单堆. 在两个堆中,指针变量 x 和 y 都分别在内存中被分配了地址. 故经过分离合取,就可以得到“x 指向 y,且 y 指向 x”的结果.

分离合取连接词“$*$”和布尔合取连接词“$\wedge$”的区别是:$P * P \neq P$ 但 $P \wedge P = P$. 尤其 $x \mapsto v * x \mapsto v$ 是永假式,因为没有办法将同一个变量 x 所对应的存储单元,划分到两个不相交的堆中. 分离合取经常被应用到表示链表结构中. 比如用 $list(x,y)$表示一个从 x 到 y 的非循环链表,将在下文的验证实例中对其形式化定义. 那么运用分离合取连接词,就可以描述某个链表片段,后面接上一个单独的节点,甚至一个终止节点为 0($null$)的链表片段,图 2 所示:

$$list(x,t) * t \mapsto y * list(y,0).$$

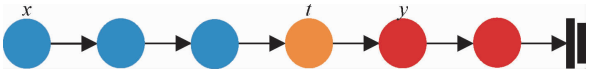

图 2　链表结构示意图

Fig. 2　A structure with a list segment

分离蕴含

$[\![p_0 -\!* p_1]\!]_a srt\ s\ h$ iff $\forall h' \cdot (h \perp h'$ and $[\![p_0]\!]_{asrt}\ s\ h')$ implies $[\![p_1]\!]_{asrt}\ s(h \cdot h')$

分离蕴含指如果有一个与当前堆 h 互不相交的拓展堆 h',可以使断言 p_0 满足. 那么断言 p_1 也可以被该拓展堆满足. 举例说明,对于

$$(x \mapsto -) * ((x \mapsto 3) -\!* Q),$$

其中 $x \mapsto -$ 指 x 已分配但不关心其具体存储的值,则该命题公式表示了 x 在当前堆中已分配的情况下,如果将其内容改为3,那么 Q 将成立,即描述了对于后置条件 Q 对于命令 $[x]=3$ 的"最弱前置条件". 这种类似 $A * (A -\!* B) \vdash B$ 的推理,被称为分离逻辑相关的演绎推理.

2.3 分离逻辑的推导规则

作为一个形式化系统的推理工具,推导规则在验证过程中起到了很重要的作用. 图3包含了分离逻辑中的部分规则,这些规则被分为描述基础数据更改操作的小公理和用于模块化推理的局部推理规则. 推理规则可以被理解为"如果横线上面的可以被推导出来,那么就可以推导出横线下的结果",而公理是已知的可满足公式.

小公理(Small Axioms)

Pointer Write (Store)

$$\{x \mapsto -\}[x] = v\{x \mapsto v\}$$

|*Pointer Read* (*Load*)

$$\{x \mapsto v\}\, y = [x]\, \{y == v \wedge x \mapsto v\}$$

Allocation

$$\{\boldsymbol{emp}\}\, x = alloc()\, \{x \mapsto -\}$$

De-Allocation

$$\{x \mapsto -\}\, free(x)\, \{\boldsymbol{emp}\}$$

局部推理规则(Local Reasoning Rules)

Frame Rule

$$\frac{\{pre\}\, code\, \{post\}}{\{pre * frame\}\, coda\, \{post * frame\}}$$

图3 分离逻辑的部分规则

Fig. 3 Partial rules of Separation logic

第一条公理表示如果 x 之前指向某个值,把其地址对应的内容改为 v 后,x 将指向新的值 v;第二条公理表示如果 x 之前存储了值 v,用变量 y 读取 x 地址所对应的内容时,y 将会暂存 v 的值. 这条公理为我们区分值在寄存器和堆中的不同描述:由于 y 只是将值 v 用作暂存,不会被分配地址空间,也就不会对堆产生更改,因此用 $y == v$ 表示,该公理假设 x 不出现在表达式 v 中;第三条公理表示如果开始时没有堆,执行分配操作后就会产生大小为1的堆;反之,第四条公理表示如果开始时是大小为1的堆,那么执行释放操作后将以空堆作为结束.

从某种意义上,小公理涵盖了它们所描述语句的关键信息. 直观来讲,这些简单的语句每次只更新或访问一个内存单元,那么仅描述这个单元格发生什么就足够了,以此可以运用就地推理的策略对程序状态进行验证,即仅在上一个状态的基础上作更新. 但又因为小公理过于简单,所以它暂时不能基于系统全局的状态进行推理.

图3中的框架规则(Frame Rule)则为分离逻辑提供了从局部推导到全局推导的逻辑上的帮助. 它允许我们将推理从一个拓展到多个存储单元,因此,仅作用于一个单元的小公理便不会再是一种限制,反而成为了一种简洁直观的描述. 以上文提到的关于构造循环链表的程序为例,下面说明运用框架规则将局部推导应用到全局,假定以 r 来描述全局的其他状态,即框架公式,为直观起见将框架公式记为红色,那么对于在上文提到的关于构造循环链表的程序,就可以做如下验证:

$$\{x \mapsto 0 * y \mapsto 0 * r\},$$
$$[x] = y;$$
$$\{x \mapsto y * y \mapsto 0 * r\},$$
$$[x] = y;$$
$$\{x \mapsto y * y \mapsto x * r\}.$$

上述推导过程说明了基于小公理的就地推理,以及框架规则对局部推理到全局的拓展,可以实现对系统全局状态的推理验证. 该例子涉及到的推理方法,可以推广到若干由链表或树作为数据结构的程序中.

2.4 验证实例

通过运用上文所介绍的分离逻辑的理论基础,本节将证明链表的原地翻转和二叉树的删除这两个程序,以说明分离逻辑的表达能力以及其实际应用价值.

2.4.1 验证链表的原地翻转程序

由于描述链表的断言是一个断言的公式序列,而且其存储情况又比较复杂,因此对于该类情况的考察,需要定义一些额外的虚拟符号便于说明. 对断言序列 α 和 β,定义如下符号:

$\alpha \cdot \beta$ 表示形如序列 α 后跟随着 β 的公式序列;

$\alpha^{\dagger}$ 表示序列 α 的映象,即序列 α 中所含内容.

使用谓词 *list* $\alpha(i,j)$ 表示第一个结点由 i 指向,最后一个结点由 j 指向的链表片段 α,其存储状态如图 4 所示.

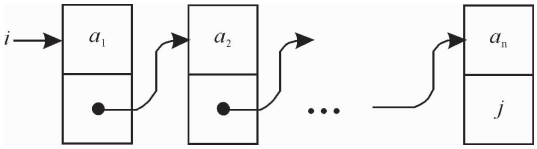

图 4　链表 *list* $\alpha(i,j)$ 示意图

Fig. 4　The semantics of *list* $\alpha(i,j)$

易知一个完整的链表应形如 *list* $\alpha(i,nil)$,*nil* 代表空结点. 根据 *list* $\alpha(i,j)$ 的结构可以直接定义断言公式如下:

$$list\ a\cdot\alpha(i,k)\overset{def}{=}\exists j.\ i\mapsto a,j*list\ \alpha(j,k),$$

以及归纳如下的基础性质:

$$list(\mathbf{nil},\mathbf{nil})\Leftrightarrow\mathbf{emp};$$

$$list\ a(i,j)\Leftrightarrow i\mapsto a,\ j;$$

$$list\ \alpha\cdot\beta(i,k)\Leftrightarrow\exists j.\ list\ \alpha(i,\ j)*list\ \beta(j,k).$$

下面演示如何使用分离逻辑,对原地反转单向链表的程序段,进行存储情况的分析以及部分正确性证明.

例:单向链表原地反转的程序 REV,代码如下:

$j:=\mathbf{nil};$

while $(i\neq\mathbf{nil})$;

do $\{k:=[i+1];[i+1]:=j;j:=i;i:=k\}$.

需证:程序开始时,链表头结点由 i 指向;当程序段执行完成后,由 j 指向反转链表的头结点. 即 $\{list\ \alpha_0(i,\mathbf{nil})\}$ REV $\{\exists\beta.\ list\ \beta(j,\mathbf{nil})\wedge\alpha_0^{\dagger}=\beta\}$.

证明

首先确定程序的循环不变式为 $\{\exists\alpha,\beta.\ (list\ \alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\}$.

(1) $\{list\ \alpha_0(i,\mathbf{nil})\}$　　前置条件

(2) $j:=\mathbf{nil};$

(3) $\{\exists\alpha,\beta.\ (list\ \alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge j:=\mathbf{nil}\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\}$

由(1)、(2)可得. 此时 $\beta=\mathbf{nil}$

(4) $\{i\neq\mathbf{nil}\}$　　由(1)可得

(5) $\{\exists\alpha,\beta.\ (list\ \alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\wedge j:=\mathbf{nil}\wedge i\neq\mathbf{nil}\}$　　由(3)、(4)可得

(6) $\{\exists\alpha,\beta.\ (list\ \alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\}$　　由(5)可得

(7) $\{\exists a,\alpha,\beta.\ (list\ a\cdot\alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=(a\cdot\alpha^{\dagger})\cdot\beta\}$　　由(6)可得

(8) $\{\exists a,\alpha,\beta,k.\ (i\mapsto a,k*list\ \alpha(k,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=(a\cdot\alpha^{\dagger})\cdot\beta\}$

由(7)可得. a 为 *list* α 头结点

(9) $k:=[i+1];$　　循环体第 1 条语句

(10) $\{\exists a,\alpha,\beta.\ (i\mapsto a,k*list\ \alpha(k,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=(a\cdot\alpha^{\dagger})\cdot\beta\}$

(11) $[i+1]:=j;$　　循环体第 2 条语句

(12) $\{\exists a,\alpha,\beta.\ (list\ \alpha(k,\mathbf{nil})*(i\mapsto a,k)*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=(a\cdot\alpha^{\dagger})\cdot\beta\}$

(13) $\{\exists a,\alpha,\beta.\ (list\ \alpha(k,\mathbf{nil})*list\ a\cdot\beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot a\cdot\beta\}$　　由(12)可得

(14) $\{\exists\alpha,\beta.\ (list\ \alpha(k,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\}$　　由(13)可得

(15) $j:=i;i:=k;$　　循环体其他语句

(16) $\{\exists\alpha,\beta.\ (list\ \alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\}$　　由(14)、(15)可得

(17) ***while*** $(i\neq\mathbf{nil})$ ***do*** $\{k:=[i+1];[i+1]:=j;j:=i;i:=k\}$

(18) $\{\exists\alpha,\beta.\ (list\ \alpha(i,\mathbf{nil})*list\ \beta(j,\mathbf{nil}))\wedge\alpha_0^{\dagger}=\alpha^{\dagger}\cdot\beta\wedge i=\mathbf{nil}\}$　　退出循环

(19) $\{\exists\beta.\ list\ \beta(j,\mathbf{nil})\wedge\alpha_0^{\dagger}=\beta\wedge i=\mathbf{nil}\}$

由(18)可得

(20) $\{\exists\beta.\ list\ \beta(j,\mathbf{nil})\wedge\alpha_0^{\dagger}=\beta\}$ 由(19)可得

上述推理验证了对链表原地反转程序的规范说明,即证明了链表原地翻转程序的正确性. 下面将举例说明,运用分离逻辑证明二叉树删除实际程序的正确性.

2.4.2　验证二叉树删除实际程序

例:二叉树删除程序 *deletetree*,实际代码如下:

```
void deletetree(struct node * root){
  if(root ! = 0){
    struct node *left = root -> l, *right = root -> r;
    deletetree(left);
    deletetree(right);
    free(root);
  }}
```

需证:程序开始时,树以 *root* 为根节点;当程序段执行完成后,系统状态为空堆. 即

$\{tree(root)\}$ *deletetree*$(root)$ $\{\mathbf{emp}\}$.

证明

首先定义描述二叉树 *tree*( )的语义如下(图 5),

$$tree(root) \overset{def}{=} \text{if } root = = \textbf{\textit{nil}} \text{ then } \textbf{\textit{emp}},$$
$$\text{else } \exists xy.\ root \mapsto [l:x, r:y] * tree(x) * tree(y).$$

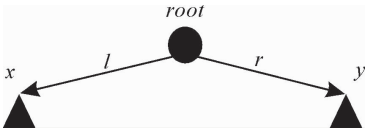

图5 $root \mapsto [l:x, r:y]$的示意图

Fig. 5 The semantics of $root \mapsto [l:x, r:y]$

这里使用谓词 $root \mapsto [l:x, r:y]$ 来描述具有 l 和 r 字段的对应的内容. 下面用分离逻辑证明删除二叉树的操作,注意使用红色标记框架公式.

$\{root \mapsto [l: left, r: right] * tree(left) * tree(right)\}$

deletetree(left)

$\{root \mapsto [l:left, r:right] * \textbf{\textit{emp}} * tree(right)\}$

deletetree(right)

$\{root \mapsto [l:left, r:right] * \textbf{\textit{emp}} * \textbf{\textit{emp}}\}$

free(root)

$\{\textbf{\textit{emp}} * \textbf{\textit{emp}} * \textbf{\textit{emp}}\}$

$\{\textbf{\textit{emp}}\}$

程序证明中的初始条件是对 $tree(root)$ 中 $root \neq 0$ 分支的语义展开. 证明步骤按照算法的直观描述:首先递归调用删除左子树,其次删除右子树,最后一个语句删除根节点. 在该证明过程中,每一步的推导都是对整体的归纳证明,并且由于框架规则,未被递归调用函数所影响的存储空间保持不变. 由此可见基于局部推理,可以简单且直观地对程序进行证明. 框架规则提供了将局部推理扩展到了全局的方法,因此,仅需对命令所改变的内容进行验证.

3 分离逻辑的扩展

3.1 并发分离逻辑

并发分离逻辑是目前较为热门的研究方向之一. 2007 年,O′Hearn[15] 提出了并发分离逻辑(concurrent separation logic, 简称 CSL),与此同时,Brookes[16] 则对 CSL 做了详尽的理论分析,给出了 CSL 的一个基于迹模型(trace model)的指称语义. Brookes 和 O′Hearn 由于对并行分离逻辑的开创性贡献而获得了 2016 年哥德尔奖. 本节将主要描述并发分离逻辑涉及到的基本想法.

对于验证完全不共享存储空间的并行程序,分离逻辑提供了一个可行的规则如下:

$$\frac{\{pre_1\}\, process_1\, \{post_1\} \quad \{pre_2\}\, process_2\, \{post_2\}}{\{pre_1 * pre_2\}\, process_1 \parallel process_2\, \{post_1 * post_2\}}.$$

该规则主要依赖于每个进程完全独立的推导,即进程间不操作共享的存储空间. 由此,对于上文提到的构建循环链表的例子就可以构建证明如下.

$$\frac{\{x \mapsto 0\}[x] = y \{x \mapsto y\} \quad \{y \mapsto 0\}[y] = x \{y \mapsto x\}}{\{y \mapsto 0 * x \mapsto 0\}[x] = y \parallel [y] = x \{(x \mapsto y) * (y \mapsto x)\}}.$$

易知 $[x] = y$ 和 $[y] = x$ 不会对共享的存储空间进行操作,因此,该推导可以一步得出最终结果.

为了灵活起见,对于带有共享内存操作的并行程序,O′Hearn 使用了带有拥有权的假设,该假设指任何代码片段只能访问当前其所"拥有的"状态. 下面举例说明该思想,对于如下带有初始化和资源定义的形式程序:

$init$;

$resourcer_1(variablelist), \cdots, resourcer_m(variablelist)$

$C_1 \parallel \cdots \parallel C_n$,

通过引入条件临界语句,用 with r then B do C endwith 来描述代码段 C,只有在其所处进程当前拥有资源 r,并且条件 B 为真的时候才可以执行. 因此就可以将共享空间中的变量,约束在其所属的进程中进行单独的推理验证,其推理规则可表示如下:

$$\frac{(\{P * RI_r) \wedge B\}\, C\, \{Q * RI_r\}}{\{P\} \text{ with } r \text{ then } B \text{ do } C \text{ endwith } \{Q\}}.$$

这里的想法是:代码 C 当前所处状态 P,且拥有资源 r 所处的状态(资源不变式 RI_r),并且可以访问这些共享状态,那么程序就以 Q 状态结束执行,且保证资源不变式仍然成立. 这条规则描述了资源共享状态拥有权的迁移. 以此并行分离逻辑可以对上述程序定义如下的推理规则:

$$\frac{\{P\}\, init\, \{RI_{ri} * \cdots * RI_{rm} * P'\} \quad \{P'\}\, C_1 \parallel \cdots \parallel C_n\, \{Q\}}{\{P\} \left(\begin{array}{c} init; \\ resourcer_1(variablelist), \cdots, resourcer_m(variablelist) \\ C_1 \parallel \cdots \parallel C_n \end{array} \right) \{RI_{ri} * \cdots * RI_{rm} * Q\}},$$

其中,$RI_{ri} * \cdots * RI_{rm}$为描述每个资源的不变式,该规则表示了对于给定的前置条件 P,经过初始化程序可建立若干资源不变式 $RI_{ri} * \cdots * RI_{rm}$,以及中间状态 P',该状态执行 $C_1 \parallel \cdots \parallel C_n$ 程序后,须满足后置条件 Q.

不过目前并发分离逻辑系统,对于并发程序的验证还需人工提供的很深入的机制,并且需引入很多共享状态使用协议、幽灵状态等作为辅助[17],因此,最终实际应用还有很长的距离.

3.2　符号执行的描述

近几年陆续产生了很多使用分离逻辑的同类型逻辑系统."符号堆"(Symbolic Heaps)是描述分离逻辑符号执行很常用的方法[18],其语法形式如下所示

$$\Delta ::= \Pi \wedge \Sigma,$$
$$H ::= \exists \vec{X}. \Delta.$$

其中 Δ 中的 Π 和 Σ 分别表示"与堆无关的公式"和"受堆限制的公式". Δ 称为无量词的符号堆公式,H 即为符号堆公式. 而且为了便于描述符号执行,符号堆中使用 $E_0 \mapsto [f_1 : E_1, \cdots, f_k : E_k]$ 的断言形式,通过标号 f_0 划分了讨论的领域,以上文所提到的

$$root \mapsto [l : left, r : right] * tree(left) * tree(right)$$

来表示根为 *root*,左子为 *left*,右子为 *right* 的二叉树,就是运用了这种公式形式. 逻辑系统通过引入符号执行,能够提高就地推导的效率. 作为第一个分离逻辑的验证工具,Berdine 等[19]就引入了符号堆,通过给定前置/后置条件和循环不变量,Smallfoot 可以构造所需的证明序列,也使得其的验证对象从仅限单堆、链表、树的断言形式,拓展到包括对数组及算术等更多的验证方向. 近年来基于分离逻辑的断言语言也大多采用了符号堆模型. 数组分离逻辑(Array Separation Logic)[20],就是符号堆分离逻辑的一种变体,其验证的数据结构包含指针和数组.

4　辅助证明方法

4.1　分离逻辑的 Coq 实现

由于逻辑推导过程的高成本降低了其实用价值,因此,近年来很多逻辑都会实现工具进行辅助证明. Coq 作为基于高阶逻辑(Church's higher-order logic)的辅助证明工具,被广泛应用到以分离逻辑为基础的推理系统中. Coq 由法国国家信息研究所(INRIA)开发,以归纳演算为理论基础,用户可通过归纳定义进行对定理或程序的构造性证明. 相比 Isabelle/HOL[21]等同类型的辅助证明工具,Coq 能精确表示程序设计语言的语法和语义,并且可以从证明中提取程序,这使得 Coq 更便于构建计算机程序相关证明系统. 如由 134 行 C 语言代码构成的 OpenSSL 中的 HMAC 认证程序[22],就由 Beringer 等基于分离逻辑的思想,以 2 832 行 Coq 代码实现了对该程序的正确性验证. 另一个更大规模的例子是对 FSCQ 文件系统的证明[23],相比较该程序的 3 千行源码,研究人员对该程序的代码抽象和证明过程约占用了 31 000 行 Coq 代码. 虽然消耗了大量的人力物力,但是实验表明该工具大大节省了验证成本,并且实现了对该文件系统的交互式证明.

4.2　双向诱导推理方法

双向诱导推理方法(Bi-Abduction)是支持对系统可拓展证明的新成果. 可拓展证明是指对于百万级别的地址空间,所需引入模块化分析的思想. 现有的证明系统,没有办法描述如此巨大的全局系统状态. 这就要求逻辑系统能够对局部代码进行分析,其中会涉及到包括代码未完成,甚至未知系统当前状态等很多技术难点. 理想的模块化推理技术,可以接受没有经过任何人工注释的代码,自动生成前置/后置规范说明,并将这些规范依次结合,最终实现对整体程序的验证[14]. 这种分析过程是组合式的,因为每一步推理都可以在不知道前提的条件下进行,即每一步都是独立的. Distefano 等[24]在模块化分析的拼接上有了新的突破,构建了双向诱导推理(Bi-Abduction)方法,该方法支持组合的形状分析(Compositional Shape Analysis)技术,能对前置/后置条件进行双向诱导推理,该方法形式如下:

$$A * ? \ anti\text{-}frame \vdash B * \ ? \ frame,$$

其中⊢读作蕴含,该式的意思是指,给定分离逻辑公式 A 和 B,*anti-frame* 和 *frame* 是未知且需要进行推理的. 通过双向诱导推理,可以通过诱导得出前置条件以实现自动化的局部推理,并可以运用框架规则来保证规范说明是最精简的. 下面举例说明双向诱导推理的思想,继续采用上文提到的循环链表的例子,但将系统的前置状态改为 *emp*,以此来说明该方法推导代码片段前置/后置条件的

过程.

当前执行状态：$\{\boldsymbol{emp}\}\ [x]=y;[y]=x\ \{???\}$,

双向诱导问题：$\boldsymbol{emp} * ? \ anti\text{-}frame \vdash (x \mapsto -) * ? \ frame$,

解决方案：$? \ anti\text{-}frame = x \mapsto -$；$? \ frame = \boldsymbol{emp}$.

之后,用 *frame* 来消除前置条件中的部分公式,并且根据小公理对当前状态进行推导,以此更新当前的执行状态.

当前执行状态：$\{x \mapsto -\}\ [x]=y\{x \mapsto y\}[y]=x\{???\}$,

双向诱导问题：$(x \mapsto y) * ? \ anti\text{-}frame \vdash (y \mapsto -) * ? \ frame$,

解决方案：$? \ anti\text{-}frame = y \mapsto -$；$? \ frame = x \mapsto y$.

这时得到的 $y \mapsto -$ 是 $[y]=x$ 的前置条件,由框架规则可知,$[x]=y$ 并不会对状态 $y \mapsto -$ 产生更改,因此,可以将其移至整体的前置条件中,并且由小公理可以得到的结果,也可以由框架规则移至整体的后置断言,因此便构成了如下推理：

$$\{(x \mapsto -) * (y \mapsto -)\},$$
$$[x]=y,$$
$$[y]=x,$$
$$\{(x \mapsto y) * (y \mapsto x)\}.$$

由此,就可以在未知当前的系统状态下,对指定代码片段进行分析验证. 这样就可以仅提取百万地址级别的巨大程序中,对想要验证的代码进行模块化分析,以此拓展到对全部程序的验证.

5 总 结

分离逻辑的理念在2001年初被提出,由于其对存储空间的描述更为细致,分析过程更为全面,因此也体现了它具有更高的应用价值. 从该理念的提出到本文写作完成,也不过经历了20年的时间,这期间围绕该模型的相关研究蓬勃发展,已经从验证操作动态内存程序的形式化推理系统,发展到如今在工具的帮助下,能够自动实现对未完成代码以及代码片段进行分析验证,甚至不用额外提供待验证代码执行前的系统状态.

然而,目前运用分离逻辑对计算机程序的验证过程还涉及到诸多困难和挑战,例如对并发程序的验证目前仍需提供大量的人工机制、对云存储等巨大复杂的存储结构没有简洁的方法描述,以及很难融合人工智能的相关成果等. 因此,将分离逻辑真正应用到工商业的实际开发中还有很长的一段路要走.

参考文献：

[1] 王戟, 詹乃军,et al. 形式化方法概貌[J]. 软件学报,2019, 30(1):33-61.

[2] Turing A M. Checking a large routine[C]//Report of a Conference on High-Speed Automatic Calculating Machines. University Mathematical Laboratory, Cambridge University, 1949: 67-69.

[3] Floyd R W. Assigning meanings to programs[C]//Proceedings of the Symposium on Applied Mathematics. J. T. Schwartz, ed. AMS, 1967: 19-32.

[4] Hoare C A R. An axiomatic basis for computer programming[J]. Communication of the ACM 12, 1969(10): 576-580.

[5] O'Hearn P W, Reynolds J C, Yang H, et al. Local reasoning about programs that alter data structures[C]//Computer Science Logic, 2001: 1-19.

[6] Reynolds J. Separation logic: A logic for shared mutable data structures[C]//IEEE Symposium on Logic in Computer Science. IEEE, 2002: 55-74.

[7] Ishtiaq S, O'Hearn P W. BI as an assertion language for mutable data structures[J]. ACM SIGPLAN Notices, 2001, 36(3):14-26.

[8] David P, Spring J M, O'Hearn Peter. Why separation logic works[J]. Philosophy & Technology, 2018(12):1-34.

[9] Yang H. An example of local reasoning in BI pointer logic: The schorr-waite graph marking algorithm[C]//Proceedings of the 1st Workshop on Semantics' Program Analysis' and Computing Environments for Memory Management, 2001:41-68.

[10] Birkedal L, Torpsmith N, Reynolds J C, et al. Local reasoning about a copying garbage collector[J]. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2004, 39(1): 220-231.

[11] Yang H, Lee O, Berdine J, et al. Scalable shape analysis for systems code[C]//Proceedings of CAV, 2008: 385-398.

[12] Berdine J, Cook B, Ishtiaq S. SLAyer: Memory safety for systems-level code[C]//Proceedings of CAV, 2011: 178-183.
[13] Yang H, O'Hearn P W. A semantic basis for local reasoning[C]//Foundations of Software Science and Computation Structure, 2002: 402-416.
[14] O'Hearn P W. Separation logic[J]. Communications of the ACM, 2019,62(2): 86-95.
[15] O'Hearn P W. Resources, concurrency, and local reasoning[J]. Theoretical Computer Science, 2007, 375(1-3): 271-307.
[16] Brookes S. A semantics for concurrent separation logic[J]. Theoretical Computer Science, 2007, 375(1-3): 227-270.
[17] 秦胜潮,许智武,明仲. 基于分离逻辑的程序验证研究综述[J]. 软件学报, 2017(8): 2010-2025.
[18] Berdine J, Calcagno C, O'Hearn P W. Symbolic execution with separation logic[C]//Asian Symposium on Programming Languages and Systems, 2005: 52-68.
[19] Berdine J, Calcagno C, O'Hearn P W. Smallfoot: Modular automatic assertion checking with separation logic[C]//LNCS FMCO, 2005(4111): 115-137.
[20] Brotherston J, Gorogiannis N, Kanovich M. Biabduction (and related problems) in array separation logic[J]. 2017(26): 472-490.
[21] Wenzel M, Paulson L C, Nipkow T. The isabelle framework[C]//Theorem Proving in Higher Order Logics, 2008:33-38.
[22] Biering B, Birkedal L, Torp-Smith N. BI-hyperdoctrines, higher-order separation logic, and abstraction[J]. ACM TOPLAS, 2007,29(5):24.
[23] Chen H, Ziegler F, Chajed T, et al. Using Crash Hoare logic for certifying the FSCQ file system[C]//Proceedings of SOSP, 2015: 18-37.
[24] Calcagno C, Distefano D, Yang H. Compositional shape analysis by means of Bi-Abduction[J]. Journal of the ACM, 2011, 58(6):1-66.

The technical foundation and research advances of separation logic

WANG Han-pin[1,2], *ZHANG Bo-wen*[2]

(1. School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou 510006, China; 2. a. Key Laboratory of High Confidence Software Technologies (MOE), b. School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

Abstract: With the increasing demand for information security, trusted computing is becoming a new research hotspot in the theoretical system of computer software. Using precise mathematical logic inference, researchers can analyze and verify the correctness of computer programs on a mathematical level. As a new method of reasoning the sharing operational storage programs, Separation Logic has gradually developed into one of the mainstream verification technologies for scalable programs. From the technical point of view, this paper introduces the research background, theoretical basis and verification principle of separation logic as well as its abstract description of storage space in detail. In order to illustrate the derivation strategy and practicability of separation logic, this paper also introduces the actual program as an example. In particular, this paper combines Bi-abduction, which is used as the theoretical basis of program automation verification based on the separation logic, to illustrates the verification process of the method for the cyclic link table program fragment. We take this as an example to demonstrate the feasibility of separating logic to verify correctness of the incomplete code and even code fragment in the event of unknown current system states. In addition, this article outlines a number of related extensions based on separation logic. With that, we can introduce the separation logic from technical details to the overall research situation.

Key words: separation logic; formal method; Bi-abduction; program verification

【责任编辑: 周　全】