软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2264−2287 [doi: 10.13328/j.cnki.jos.006581]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种基于分离逻辑的块云存储系统验证工具*

张博闻[1,3], 金 钊[1,3], 王捍贫[1,2,3], 曹永知[1,3]

[1](北京大学 计算机学院, 北京 100871)

[2](广州大学 计算机科学与网络工程学院, 广东 广州 510006)

[3](高可信软件技术教育部重点实验室(北京大学), 北京 100871)

通信作者: 王捍贫, E-mail: whpxhy@pku.edu.cn

摘 要: 云存储技术目前被广泛应用于人们的生产与生活中. 验证云存储系统中管理程序的正确性, 能够有效地提高整个系统的可靠性. 块云存储系统(CBS)具有最接近底层的存储架构. 运用交互式定理证明器 Coq, 实现了一种辅助验证工具, 用于分析和验证 CBS 中管理程序的正确性. 基于分离逻辑的思想, 对工具中证明系统的实现主要包括:首先, 将CBS抽象为两层堆结构, 定义建模语言形式化表示CBS的状态和管理程序; 其次, 定义描述CBS状态性质的堆谓词, 并说明堆谓词间的逻辑关系; 最后, 定义描述程序行为的 CBS 分离逻辑三元组, 以及制定验证三元组所需的推理规则. 此外, 还引入了几个证明实例, 以此展示工具对实际CBS管理程序表示和推理的能力.

关键词: 分离逻辑; 交互式定理证明器; 块云存储系统; 形式化验证; Coq

中图法分类号: TP311

中文引用格式: 张博闻, 金钊, 王捍贫, 曹永知. 一种基于分离逻辑的块云存储系统验证工具. 软件学报, 2022, 33(6): 2264–2287. http://www.jos.org.cn/1000-9825/6581.htm

英文引用格式: Zhang BW, Jin Z, Wang HP, Cao YZ. Tool for Verifying Cloud Block Storage Based on Separation Logic. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2264−2287 (in Chinese). http://www.jos.org.cn/1000-9825/6581.htm

Tool for Verifying Cloud Block Storage Based on Separation Logic

ZHANG Bo-Wen[1,3], JIN Zhao[1,3], WANG Han-Pin[1,2,3], CAO Yong-Zhi[1,3]

[1](School of Computer Science, Peking University, Beijing 100871, China)

[2](School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou 510006, China)

[3](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

Abstract: Cloud storage is now widely used in production and people's life. Verifying the correctness of hypervisors in cloud storage can effectively improve the reliability of the whole system. Cloud block storage (CBS) has the closest storage architecture to the bottom layer. In this study, a tool is implemented for analyzing and verifying the correctness of hypervisors in CBS, by using the interactive theorem prover Coq. Based on separation logic, the implementation of the proof system in the tool mainly consists: First, a modeling language is defined to abstract the CBS into a two-tier structure, and to formally represent the CBS state and the hypervisor; second, several predicates are defined to describe the state properties of the CBS, and the logical relationships between predicates are illustrated; finally, a separation logic triple for CBS is defined to describe the behavior of a program, and the reasoning rules required to verify the triples are stated. In addition, several proof examples are introduced in this study, to present the tool's ability to represent and reason about the real hypervisor in CBS.

Key words: separation logic; interactive theorem prover; cloud block storage; formal verification; Coq

近年来, 随着互联网技术发展日新月异, 云服务也取得了突破性的进展. 据市场调研报告表明, 2020 年云

* 基金项目: 国家科技攻关计划(2018YFB1003904, 2018YFC1314200); 国家自然科学基金(61772035, 61972005, 61932001)
本文由"定理证明理论与应用"专题特约编辑曹钦翔副教授、詹博华副研究员、赵永望教授推荐.
收稿时间: 2021-09-07; 修改时间: 2021-10-14; 采用时间: 2022-01-04; jos 在线出版时间: 2022-01-28

服务的市场规模已经达到了 3 714 亿美元[1]. 云存储系统作为整个云生态的数据基石, 扮演着重要的角色. 云存储凭着成本低、灵活度高、便于扩展等优点, 受到国内外公司的青睐, 其市场潜力被一致看好. 我国 2021 年发布的“十四五规划”中, 更将云存储的建设列为国家发展战略目标[2]. 然而, 随着云存储应用日渐频繁, 其可靠性在近年来也不断受到人们的质疑. 据统计, 每年由云存储故障导致的直接损失近 2.85 亿美元[3]. 例如 2020 年, 亚马逊云服务系统数十台服务器同时崩溃了近 5 个小时, 直接使得数千家第三方软件和平台的在线服务陷入停摆, 其原因主要是亚马逊“小幅提升了服务器容量”所致[4]. 另一方面, 尽管运营商提供了数据转移等容错机制, 但云存储依然存在许多服务故障和可靠性问题[5]. 为避免类似情况再次发生, 减少数据和财产损失, 人们有必要对云存储系统本身的可靠性进行深入、系统的研究. 这里的可靠性主要涉及到两个层面: 硬件基础设施的稳定性、数据管理程序的正确性. 管理程序与用户直接相关, 它是用于处理用户请求以及管理存储系统的一系列命令, 如创建、添加、删除等. 提高管理程序的正确性, 对提升云存储系统的可靠性十分关键. 目前, 市场上主流的云存储系统可以分为 3 类: 文件存储、对象存储和块存储[6]. 这种分类方式主要是针对系统的架构, 以及数据存储级别的不同. 块云存储系统(cloud block storage, CBS)与存储介质的交互最为直接, 可为另外两类云存储系统的实现提供接口[7]. 因此, 本文主要关注的是对块云存储系统中管理程序的正确性验证.

程序正确性一般是指输入都会产生期望的结果, 验证它的主流研究方法有两种: 测试和形式验证. 测试的方法对于大型系统很难做到整体上的覆盖. 亚马逊的云存储平台 AWS 在开发时, 编程人员就表示过测试难以分析如此复杂的分布式系统[8]. 另一方面, 形式化验证[9,10]运用数学模型, 能够分析程序是否具有严谨的正确性. 定理证明技术[11−13]是一种基于数理逻辑的形式验证方法, 通过抽象描述计算机程序和大型系统, 人们可以建模出具有推导能力的公理化系统, 以此运用形式推理来提高系统的可靠性和鲁棒性. 定理证明技术在传统存储的验证上, 出现过很多杰出的成果. 不过, 相对本地存储来讲, 云存储系统的架构更为复杂. 此外, 随着存储需求的增长, 云存储系统的规模也不固定. 诸多因素导致云存储系统的形式化验证充满挑战, 很难直接运用传统的验证技术. 这就需要对原有的方法进行拓展和创新, 目前已有的对云存储的验证工作, 主要的验证方向是多副本之间的数据一致性、存储数据的高可用性和完整性等. 而对涉及逻辑存储的讨论较少, 这里的逻辑存储涉及文件目录、块地址等定位目标数据的信息, 它们是程序执行数据调配的关键. 本文运用定理证明技术, 从逻辑存储的角度来分析验证 CBS 管理程序的正确性.

在带地址操作程序的验证领域中, 分离逻辑(separation logic)[14,15]是一个很重要的突破. 它最初由 O’Hearn 和 Reynolds 提出, 是一个基于 Hoare 逻辑[16]的形式化逻辑系统. 经历了十几年的发展, 它从一种推理共享可操作存储程序的方法, 成为目前主流的程序形式化验证技术. 其代表创始人 O’Hearn 也凭借这个工作, 在 2016 年获得了理论计算机领域最负盛名的“哥德尔奖”. 分离逻辑能够直观地验证带有复杂存储操作的程序, 它通过形如“$\{H\}t\{Q\}$”的三元组, 来描述一个程序 t 的行为. 其中, 前置断言 H 描述输入状态, 后置断言 Q 描述输出状态, 这样的三元组被称为关于 t 的一个规范[17]. 分离逻辑提倡小规模规范(small-footprint specification), 它意味着三元组涵盖的系统状态要尽可能地小, 只需提及与命令执行有关的必要部分, 而任何前置条件中没有明确描述的状态都默认保持不变. 此外, 分离逻辑局部推导(local reasoning)的特性, 可以令推理更加精确[18]. 局部推导主要基于 frame 规则, 该特性意味着, 如果一个程序能够在给定的部分状态中安全执行, 那么这个程序也可以在更大的系统状态中安全执行. 结合小规模规范和局部推导, 人们验证程序时可以只关注与操作有关的部分状态, 而无需讨论无关状态, 随后运用 frame 规则, 就可以将局部的性质推广到全局[19]. 可以看出, 分离逻辑的推理模式对验证复杂的存储系统很有帮助, 尤其是对 CBS 的验证.

为了提高理论模型的实用性和证明效率, 研究人员一般会在交互式定理证明器中实现一个验证工具, 方便对研究目标的推理. 同时, 验证工具还能够提供一个机器可校验的证明(machine checkable proof), 避免了人力验证时可能发生的疏漏[20,21]. 由此, 诸多基于分离逻辑的验证工具应运而生. Charguéraud 等人在 Coq 中实现的 CFML, 就是一个基于分离逻辑、针对 ML (meta language)函数式语言的验证工具[22−24]. CFML 具备分离逻辑的特性, 如小规模规范、局部推导等, 这些特性使得它可以支持验证链表、树、并查集等复杂的数据

结构[25,26]. 该工具引入 ML 语言中的“不可变变量”和“可变堆空间”的概念. 不可变变量是指, 变量一旦被赋值后, 就不可以再被更改. 可变堆空间是指, 只有在存储单元中的值, 才能够被改变. 这样能避免可变变量所引发的复杂情况, 也使得代码更易推理[27]. 运用这个概念, CFML 运用λ演算的形式, 相对简洁地将分离逻辑深层嵌入(deep embedding)到了 Coq 中, 实现了分离逻辑的一个简单变体. 在 Charguéraud 编写的软件基础(software foundations)系列的第 6 卷中, 前 4 章就详细介绍了关于分离逻辑的实现细节[28].

基于这本书前 4 章对分离逻辑的实现思想, 本文开发了一个针对 CBS 的验证工具. 之所以这样选择, 其原因主要有以下几点: 首先, CBS 管理程序的操作对象是存储单元, 运用不可变变量和可变堆空间的形式, 可以使我们将目光更多地聚焦在带地址的操作上. 其次, λ演算能够抽象掉程序设计语言中的无关细节, 进而能够简洁并且直观地表示文件和块的操作过程, 比如运用λ演算的归约, 可以直接表示对程序的传参. 技术上讲, 在 Coq 中对证明系统的实现, 又依赖于 λ 演算的形式.

在众多 CBS 产品中, Hadoop 分布式文件系统(Hadoop distributed file system, HDFS)[29]是最具代表性的产品之一. HDFS 具有 CBS 的典型存储方式, 即通过将文件切割为若干离散的数据块, 来实现对大型数据集的部署. HDFS 的存储架构为主从式, 主节点负责存储文件目录, 以及数据块的元数据. 这里的元数据包含块的大小, 块的位置和从属于哪个文件等信息. 从节点负责将数据块的实际内容, 存放到本地磁盘中. 目前市面上主流的 CBS 产品, 具备与 HDFS 类似的架构和存储方式. 总体来说, CBS 中管理程序对数据的调配, 都会涉及“文件-数据块-块内容”的存储逻辑, 分析和验证程序的正确性势必要考虑这个存储关系.

针对 CBS 管理程序在逻辑存储层面上的正确性, 在之前的工作中, 我们曾提出过一种建模语言来表示 CBS 的系统状态和管理程序[30]; 基于这个建模语言, 并运用分离逻辑的思想, 我们还构建过一个证明系统, 来分析验证 CBS 管理程序的行为[31]. 不过, 这些工作存在一些需要改进的地方, 比如它们专注于数据块的存储安全, 但忽略了文件的存储; 以及数据块内容只能为一个整数, 推理只能在纸笔上进行等. 本文实现的验证工具, 对这几点都做出了改进. 在工具的证明系统中, 我们额外描述了主节点的存储状态, 增加了对 CBS 文件操作的推理能力, 并且还将块内容扩充表示为一个整数序列, 提高了模型语言的表达能力. 此外, 在 Coq 中进行推理, 能够有效避免纸笔推导上的疏漏, 同时工具中的自动化脚本和策略, 也能减少一定的人力成本.

总体来说, 本文的主要工作是在交互式定理证明器 Coq 中, 实现了一个针对 CBS 的验证工具. 本文工具具备分离逻辑的关键特性, 尤其能够支持对 CBS 程序进行局部推导. 对应实际中的主从式架构, 工具将 CBS 细分为两个存储层级: 文件层、块层. 通过整合内部层级的状态和操作, 工具支持表示和验证实际 CBS 的各项数据操作. 详细来讲, 本文在 Coq 中, 基于分离逻辑实现了一个关于 CBS 的证明系统, 它涉及到构建建模语言、CBS 堆谓词、分离逻辑三元组和推理规则等环节.

(1) 制定建模语言表示 CBS 的系统状态、管理程序以及指令对状态的影响. 基于分离逻辑的思想, 本文用 CBS 堆表示一部分的 CBS 系统状态. 对应实际中的主从式架构, 本文抽象出一个两层堆结构: 文件堆和块堆, 它们分别表示“文件-块”和“块-块内容”的映射关系. CBS 堆被定义为由文件堆和块堆所组成的二元组. 此外, 工具中新引入一系列原子操作, 对应了文件和块的实际操作环节. 由原子操作组成的复合语句, 可以表示实际 CBS 的管理程序. 随后, 运用带有返回值的操作语义, 工具中定义了语法树中每个构造元素的求值规则, 来表示程序执行对系统状态的更新.

(2) 定义有关 CBS 堆的谓词, 以描述 CBS 系统状态的性质. 基于分离逻辑, 本文通过 CBS 堆的运算, 定义了一系列 CBS 堆谓词, 来描述 CBS 状态的性质. 同时, 对应两层结构, 本文还定义了内部的堆谓词, 以分别描述文件层和块层的性质. 此外, 本文制定了 CBS 堆谓词间的蕴含关系, 以支持堆谓词间的语义推导. 特别地, 我们在工具中证明了宏观上 CBS 堆谓词间的分离合取, 逻辑等价于在内部堆谓词上的分离合取, 这个等价性为证明随后的推理规则提供了基础.

(3) 定义 CBS 分离逻辑三元组来规范程序的行为, 制定推理规则来验证三元组. 针对新引入的 CBS 堆谓词, 工具定义了 CBS 分离逻辑三元组, 用于描述程序的行为. 同时, 为推理和验证三元组, 本文对于新引入的原子操作, 制定了一系列推理规则. 另外, 还按照 CBS 三元组的形式, 重写了传统分离逻

辑的推理规则. 在工具中, 我们证明了所有推理规则的可靠性. 尤其是 frame 规则的成立, 说明了工具支持对 CBS 程序的局部推导.

本文第 1 节大致说明研究方法. 首先以 HDFS 为代表, 介绍对 CBS 存储架构的抽象思想; 其次展示工具如何对 CBS 管理程序进行表示和推理. 第 2 节阐述对建模语言的定义, 主要分为 CBS 堆、语法树、推理规则等方面. 第 3 节阐述如何定义 CBS 堆谓词和蕴含关系. 第 4 节阐述 CBS 分离逻辑三元组的定义以及各项推理规则的制定. 第 5 节引入 CBS 的验证实例, 说明工具对 CBS 的推理能力和实用性. 第 6 节介绍一些相关工作. 最后一节给出总结和展望.

工具共涉及 3 325 行 Coq 代码实现, 包括 51 条定义, 242 条引理, 并提供了 7 个证明实例. 它目前被发布于 https://github.com/PKUTCS-CSS/CBSVerifi.

1 方法概览

本节介绍对 CBS 架构的抽象过程. 以 HDFS 为代表, 展示 CBS 产品的主从式存储架构, 说明引入两层堆结构的实际含义. 此外, 本节还介绍工具对 CBS 管理程序的验证方式. 以数据块的复制为例, 展示工具如何表示实际中的数据操作、如何规范程序的行为以及如何验证程序的正确性等, 以此说明工具的表达和推理能力.

1.1 抽象CBS架构

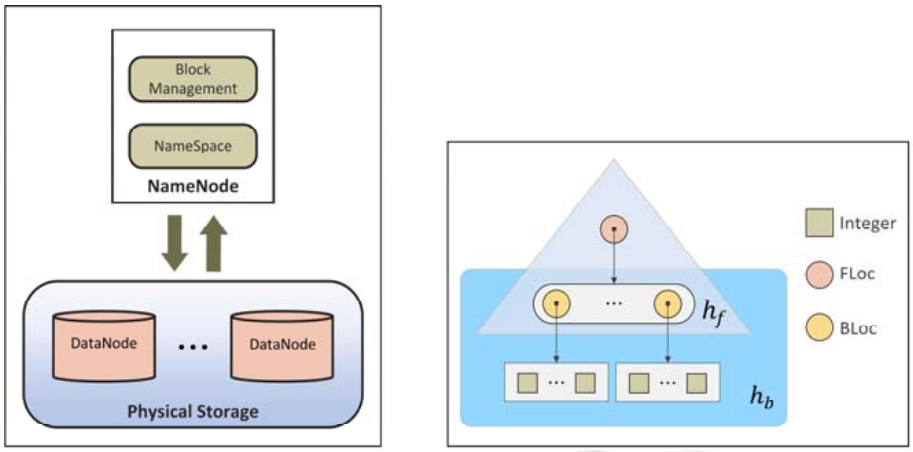
Hadoop[32]是一个支持数据密集型分布式应用程序的开源软件框架, 它的核心组件是负责数据存储的 HDFS, 以及负责数据处理的 MapReduce 并行运算框架. 作为整个 Hadoop 生态的数据基础, HDFS 维护的单个文件规模一般为 GB 甚至 PB. 2003 年, 由谷歌公司发表的论文“Google 文件系统”[33], 为目前众多块云存储平台提供了理论基础. HDFS 就是其中最具代表性的产品之一, 它具有 CBS 典型的主从式架构. 一个经典的 HDFS 集群通常由单个“名称节点(下称 NameNode)”作为主节点, 以及多个“数据节点(下称 DataNode)”作为从节点所构成, 架构示意图如图 1(a)所示[34].

NameNode 是 HDFS 的核心, 它负责维护管理数据块的命令集(block management)和文件系统(NameSpace). 前者包含操作数据块的相关命令, 后者存储着文件名、目录树、块的元数据等重要信息. 数据块的元数据涉及了块大小、块地址、文件到块的映射等. 有关集群中文件和块的任何操作, 都会相应地更新 NameNode 的文件系统, 尤其是块的元数据. DataNode 是 HDFS 的物理存储基础, 它将各个块的实际内容存放到本地磁盘. 数据块是 HDFS 存储和操作的基本单元, 大数据文件会被拆分为若干相互分离的数据块, 每个块都具有唯一的块地址, 块的内容会被落地在不同的 DataNode, 块的元数据则会被统一维护在 NameNode.

本文将 CBS 的存储架构抽象为两层: 文件层和块层, 分别对应主节点和从节点的存储层级. 也就是说, 若干数据节点将被统一抽象为块层. 我们引入了文件堆和块堆, 来表示对应层级的系统状态, 如图 1(b)所示, 其中文件堆是“文件地址-块地址序列”的有限映射, 块堆是“块地址-块内容”的有限映射. 借此, CBS 的系统状态就被表示为由文件堆和块堆组成的二元组, CBS 堆也就由这样的二元组定义.

需要注意的是, 本文的建模保留了 CBS 相对于普通文件系统的特殊性. 详细来说, 传统存储中的数据块是在物理层面上做出的划分, 因此每个块的容量都固定; 并且, 由于硬件的限制, 块数量也具有上限. 相比较而言, CBS 中的块是一个虚拟出来的存储空间, 虽然也是由一个地址索引, 但是块容量支持用户自行定义. 并且, CBS 允许随时添加节点来扩充数据中心的规模, 以存放更多的数据块. 对应地, 该模型中每个块地址在块堆中的映射结果是一个非定长的序列, 这意味着我们没有固定块的容量. 此外, 块堆的大小是弹性的, 我们也没有将块数量的上限限定为固定的常数.

为了表示 CBS 的数据处理环节, 模型语言中新引入了一系列文件和块的原子操作, 它们代表无法被中断和细分的操作环节. 通过将原子操作组成复合语句, 可以表示实际中的各项数据操作. 比如读取块和创建块的顺序执行, 就能表示对一个数据块的复制.



(a) HDFS的实际存储架构 (b) 抽象后的两层堆结构

图 1 抽象以HDFS为代表的CBS产品的存储架构

1.2 验证CBS的管理程序

下面就以复制一个数据块为例, 说明本文工具是如何表示和验证 CBS 的管理程序的. 如上文所述, 该操作可以被划分为两个原子操作: 读取块和创建块, 在工具中就以如下编码的程序表示. 其中, “Fun bk”意味着该程序需要一个参数, bget 和 bcreate 分别表示两个原子操作.

```
Definition Copy_blk: val: =
  Fun bk: =
    Let ln: =bget bk in
      bcreate ln.
```

工具对程序 t 行为的描述, 采用形如“triple t H Q”的规范, 它是针对 CBS 重新定义的 CBS 分离逻辑三元组. 这里的 H 和 Q 是三元组的前置和后置条件, 它们是描述 CBS 系统性质的堆谓词. 按照两层堆结构, 一个 CBS 堆谓词还可以被细化成“\R[Hf, Hb]”的形式, 以更精确地刻画内部状态的性质, 其中的 Hf 和 Hb 分别是描述文件层和块层状态性质的堆谓词.

工具中, 我们通过“Copy_blk bp”来表示块复制程序的调用, bp 是一个块地址, ln 是一个整数序列. 工具用如下的引理来声明它执行的正确性. 按照分离逻辑的思想, 前置条件中未提及的状态不会发生改变. 由于 Copy_blk 程序只操作目标块, 并不影响其他的数据块和文件, 因此前置条件只需描述该块即可.

```
Lemma triple_Copy_blk: forall (Hf: hfprop)(bp: bloc)(ln: listint),
  triple(Copy_blk bp)
    (\R[Hf,bp~b~>ln])        (*前置条件*)
    (fun r=>(\R[Hf,(\existsb bp',\b[r=bp'] \b* (bp'~b~>ln)\b* (bp~b~>ln))])).        (*后置条件*)
```

非形式化来说: 后置条件“(fun r=>…)”的形式用于代入程序的返回值; “bp~b~>ln”是一个块单堆, 它表明当前系统中只有一个数据块, 该块的地址为 bp 且内容为 ln; “H1\b* H2”是两个块堆谓词之间的分离合取; “\b[P]”是不依赖块层状态的纯事实(pure fact); “\existsb”是在块堆谓词层面的存在量化.

上述规范的含义为: 前置条件描述了系统中存储着一个数据块, 它的地址为 bp, 内容为 ln; 在程序执行结束之后, 后置条件描述当前系统中新存入了一个数据块, 它位于某个地址 bp′, 并且内容也为 ln. 同时, 程序的返回值是新数据块的地址, 而且程序的执行不会改变文件层的状态. 观察到, 后置条件中的“(bp'~b~>ln)\b* (bp~b~>ln)”表明两个新旧数据块之间的分离合取. 按分离逻辑的思想, 这时数据块层能够被“分离”为不相交的两部分, 这两部分可以分别满足运算符左右断言的性质. 而其中的“不相交”, 则恰恰保证了这两个新旧块的块地址有所不同.

在工具中, 运用到推理规则和堆谓词间的蕴含, 可以实现对上述规范的验证, 在 Coq 中的证明脚本如下所示.

```
Proof.
  intros.applys* triple_app_fun.                    (*初始化证明, 运用程序调用的规则*)
  applys triple_let triple_bget.ext.                (*展开 let 绑定, 运用读取块的规则, 提取返回值*)
  applys triple_conseq_frame triple_bcreate.        (*改写三元组, 运用创建块的规则*)
  rewrite* hstar_hempty_l′.                         (*改写后的前置条件能蕴含*)
  apply* himpl_hbexists.                            (*改写后的后置条件能蕴含*)
Qed.
```

对应实际中文件和数据块的操作环节, 工具可编码表示 CBS 的管理程序. 结合 CBS 的系统架构, 工具专门为 CBS 定义了分离逻辑三元组, 以描述编码程序的行为. 随后运用各种推理规则, 我们可以在 Coq 中实现对三元组的相应证明.

需要注意的是, 本文主要采用数学符号来介绍工具的实现原理. 这是考虑到, 证明系统的实现过程比较复杂, 运用代码解释的可读性较差, 而数学表示能够使行文更加简洁和直观. 我们鼓励读者访问工具的发布网站, 检查和审阅相应的源码, 以了解工具实现的具体技术细节.

2 建模语言的语法和语义

本文工具考虑的语言形式是“指令式调用值的λ演算(imperative call-by-value λ-calculus)”[27], 它与传统分离逻辑中的指令式语言略有不同. 首先, 分离逻辑语言中的栈(store), 被用于变量和表达式的求值. 而本文语言采用了 ML 语言中“不可变变量”和“可变堆空间”的理念, 这意味着, 由 let 绑定直接赋值的变量, 不允许再被修改; 只有被分配到存储空间的值, 才支持被修改. 工具中定义的替换函数可以将变量直接替换为值, 以达到与栈类似的作用. 其次, 本语言中的指令执行, 不仅会操作系统状态, 同时还会输出一个返回值, 对于删除等不需要输出执行结果的指令, 也会返回一个 unit 类型的可忽略值. 最后, 运用λ演算的归约方式, 该语言能够直接表示对程序的传参调用以及返回值的代入.

本节介绍工具中建模语言的构建, 主要涉及 CBS 状态、语法树和求值规则的定义和解释. 按照实际架构, 工具运用两层堆结构表示 CBS 的系统状态. 一个 CBS 堆, 表示了一部分的 CBS 状态, 由文件堆和块堆组合定义. 此外, 工具中新引入了一系列原子操作, 用来表示实际中文件和块的处理环节. 随后, 运用与原子操作无关的项(如 let 绑定、条件语句等), 就可以构造出复合语句来表示各种 CBS 管理程序. 此外, 工具运用操作语义, 对每个语法元素都制定了求值规则, 以此表明程序执行对 CBS 堆的更新和相应的返回值.

简单起见, 验证工具中涉及到的值均为自然数, 这样能够避免引入浮点数所导致的一系列琐碎问题. 同时, 这也是因为我们更关心对数据管理上的验证, 而非算术层面上的性质.

2.1 CBS堆

本文将 CBS 细分为两层结构: 文件层和块层. 为了表示 CBS 的状态, 首先需要表示各存储层级的状态. 下面, 令 *floc* 为文件地址类型, *bloc* 为块地址类型, 它们可以通过自然数来实现. 令 list(*bloc*)表示块的地址序列, list(*int*)表示整数序列, 令 fmap α β表示从α到β的有限映射类型.

一个文件堆表示文件层的一部分状态, 被定义为文件地址到块地址序列的有限映射. 同理, 块堆被定义为块地址到整数序列的有限映射. 其中的“有限映射”, 保证了新的地址总会存在.

定义 2.1(文件堆). 文件堆 heapf 的类型为“ fmap *floc* list(*bloc*) ”.

定义 2.2(块堆). 块堆 heapb 的类型为“ fmap *bloc* list(*int*) ”.

随后, 令 h_f 和 h_b 分别表示 heapf 和 heapb 类型的元变量. 对于堆之间的运算, 以 $h_f \perp h'_f$ 表示两个文件堆不相交, 即没有一个文件地址同时属于 h_f 和 h'_f 用 $h_f \uplus h'_f$ 表示合并两个不相交的文件堆. 注意, 关于文件堆

的两个运算符约定, 同样也适用于块堆.

一个 CBS 堆, 表示一部分的 CBS 系统状态, 由文件堆和块堆组合的二元组表示.

定义 2.3(CBS 堆). CBS 堆 heap 的类型为“heapf × heapb”.

随后, 令 h 表示一个 CBS 堆, 通过语法可知, h 能够被细分为 (h_f, h_b) 的形式, 两个 CBS 堆之间的运算就采用了细化后的形式来描述. 我们来考虑两个 CBS 堆 h_1 和 h_2, 假设它们被细分后形如 (h_f, h_b) 和 (h'_f, h'_b). 随后, 我们将两个堆不相交记为 $h_1 \perp h_2$, 它的定义为 $(h_f \perp h'_f) \wedge (h_b \perp h'_b)$; 两个堆的合并被记为 $h_1 \uplus h_2$, 它的定义为 $\big((h_f \uplus h'_f), (h_b \uplus h'_b)\big)$. 在实际推理中, 细化后的 CBS 堆能够更有效地表示原子操作的求值.

2.2 语 法

本工具的语法范畴涉及到文件原子操作 *fprim*、块原子操作 *bprim*、值 v 和项 t. 对应两层结构, 本文新引入了文件和块的原子操作. 同时, 还将有关文件和块的值, 相应地引入到 v 之中. 与原子操作无关的项 t, 则沿用了之前工具中的语法形式[27].

定义 2.4(建模语言的语法).

$$
\begin{aligned}
\mathit{fprim} &:= \text{fcreate} \mid \text{fget} \mid \text{fdelete} \mid \text{attach} \mid \text{nthblk} \mid \text{fsize} \mid \text{fset} \\
\mathit{bprim} &:= \text{bcreate} \mid \text{bget} \mid \text{bdelete} \mid \text{append} \mid \text{bsize} \\
v &:= tt \mid n \mid b \mid f \mid be \mid ln \mid lb \\
&\quad \mid \mathit{fprim} \mid \mathit{bprim} \mid \hat{\lambda} x.t \mid \hat{\mu} F.\lambda x.t \\
t &:= v \mid x \mid (t\ t) \mid \text{if } t \text{ then } t \text{ else } t \\
&\quad \mid \text{let } x = t \text{ in } t \mid t_1; t_2 \mid \lambda x.t \mid \mu F.\lambda x.t
\end{aligned}
$$

文件原子操作中, 涵盖的文件操作有: 创建、读取块序列、删除、追加块、索引第 n 个块、统计文件的数据块数量, 以及更新第 n 个块. 同理, 块原子操作中, 涵盖的块操作有: 创建、读取、删除、追加和计算块大小.

值涵盖了 unit 类型的无意义值 tt、整数 n、块地址 b、文件地址 f、布尔值 be、整数序列 ln、块地址序列 lb、文件原子操作 *fprim*、块原子操作 *bprim*、非递归程序和递归程序. 尖角符号表示这里的程序对于值是闭包的, $\hat{\mu} F.\lambda x.t$ 中的 F 是进行递归调用时的程序名.

项涵盖了值、变量、程序调用、条件语句、let 绑定、顺序语句、非递归和递归程序. 注意, 这里的 $\lambda x.t$ 和 $\mu F.\lambda x.t$ 是作为项的非递归和递归程序, 它们并不带有尖角符号.

下面结合上述语法树的对应 Coq 实现, 来理解工具对 CBS 程序的编码方式. 为了支持带有返回值的指令执行, 文件和块的原子操作, 以及各种程序都可以作为值. 因此在工具的语法树中, 我们也直接将文件和块的原子操作分别命名为 fval 和 bval(见下表左半部分). 工具用 trm_app 表示程序调用. 那么, 如果严格按照类型系统, 对于一个地址为 bp 的数据块, 对其内容的读取应编码为“trm_app((trm_val(val_bval bval_get))(trm_val (val_bloc bp)))”. 可见, 这样编码程序相当繁琐. 运用 Coq 中的强制类型转换技术进行处理, 我们简化了编码中对类型的标注, 支持将上述的调用语句在工具中直接编码为“(bval_get bp)”, 这同样也对应了文中 $(t\ t)$ 的数学表示.

此外, 对于递归和非递归程序来讲, 将它们在项和值类型中做出区分, 是为了能够表示带有多个参数的程序. 工具中, val_fun 是值类型的非递归程序, 而 trm_fun 是项类型的 $\lambda x.t$. 我们用“Fun x1: =t”来编码一个值类型的单参数非递归程序, 即“val_fun x1 t”, 注意, 这里 t 的类型是项. 那么, 编码需要两个参数的程序“Fun x1 x2: =t”, 按照 Coq 中的类型系统, 应为“val_fun x1(trm_fun x2 t)”的形式. 关于多参数递归程序的处理方式同理.

```
Inductive fval: Type: =
  |                  : fval
  fval_create        : fval
  |                  : fval
  fval_attach
  | fval_fsize
  | …

Inductive bval: Type: =
  | bval_create      :
  |                  bval
  bval_append        :
  | bval_get         bval
  | …                :
                     bval

Inductive val: Type: =
  | val_unit         : val
  | val_bloc         : bloc->val
  | val_listint      : list int->val
  |                  : list bloc->val
  val_listbloc       : bval->val
  | val_bval         : var->trm->val
  | val_fun
  | …

with trm: Type: =
  | trm_val          : val->trm
  | trm_app          : trm->trm ->trm
  | trm_fun          : var->trm->trm
  | …
```

2.3 语 义

工具中, 各个语法元素的求值依赖于当前全局的系统状态, 注意是全局状态. 下面, 我们用 s 来表示一个 heap 类型的元变量, 它描述全局的 CBS 存储状态. 注意区分, s 与 h 不同, 后者只会描述部分状态. 建模语言的求值规则, 由带返回值的操作语义定义, 它是形如"$t/s \Downarrow v/s'$"的格局, 该式描述了: 在全局状态 s 下执行项 t, 执行完成后系统的终止状态为 s', 项的返回值为 v. 语义的定义分为两部分: 首先是项的求值规则, 它们与原子操作无关, 主要表示语句的执行方式; 其次是文件和块的原子操作的求值规则, 它们表示各原子操作对系统状态的更新细节.

定义 2.5(项的求值规则).

$$\frac{}{v/s \Downarrow v/s} \qquad \frac{}{(\lambda x.t)/s \Downarrow (\hat{\lambda} x.t)/s} \qquad \frac{}{(\mu F.\lambda x.t)/s \Downarrow (\hat{\mu} F.\lambda x.t)/s}$$

$$\frac{([v_1/x]t)/s \Downarrow v/s'}{((\hat{\lambda} x.t)\ v_1)/s \Downarrow v/s'} \qquad \frac{v_1 = \hat{\mu} F.\lambda x.t \qquad ([v_2/x][v_1/F]t)/s \Downarrow v/s'}{(v_1\ v_2)/s \Downarrow v/s'}$$

$$\frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]t_2)/s' \Downarrow v/s''}{(\text{let}\ x = t_1\ \text{in}\ t_2)/s \Downarrow v/s''} \qquad \frac{t_1/s \Downarrow v'/s' \qquad t_2/s' \Downarrow v/s''}{(t_1; t_2)/s \Downarrow v/s''}$$

$$\frac{\text{if}\ be\ \text{then}\ (t_1/s \Downarrow v/s')\ \text{else}\ (t_2/s \Downarrow v/s')}{(\text{if}\ be\ \text{then}\ t_1\ \text{else}\ t_2)/s \Downarrow v/s'}$$

项的求值规则, 只是对之前工具中的语法元素, 在新系统状态下的重新解释. 第 1 行的规则说明: 值、非递归和递归程序的求值, 就是它们的本身; 第 2 行说明程序调用的求值规则, 它们依赖于替换函数"$[v/x]t$". 替换函数"$[v/x]t$"可将项 t 中所有的 x 都替换为 v. 调用非递归程序是直接在 t 中用参数替换变量, 递归程序在此基础上还额外引入了上一次执行的返回值, 来替换项中的程序名 F, 以此避免了递归调用时可能发生的混淆; 第 3 行说明两种顺序语句的求值规则, 它们的不同之处在于: let 绑定会将执行中间的返回值带入到后续指令中, 而单纯的顺序语句则不会; 最后是条件语句的求值规则, 它直接运用 Coq 的条件语句来定义.

随后, 我们重点针对新引入的文件和块原子操作, 分别定义它们的求值规则. 考虑到文件和块原子操作只会发生在相应层级, 因此需要细化表示 CBS 状态. 同理, 我们引入 heapf 类型的元变量 s_f 和 heapb 类型的元变量 s_b, 用 s_f 和 s_b 分别表示文件层和块层的全局状态, 并由 (s_f, s_b) 来细化表示整个 CBS 的全局状态. 原子操作的求值规则, 也就可以通过细化后的格局"$t/(s_f, s_b) \Downarrow v/(s_f', s_b')$"来定义.

原子操作的语义与映射更新和列表操作有关. 为方便说明, 首先约定一些符号表示. 关于映射更新, 对任意的有限映射 m, 下文用 $\mathrm{dom}(m)$ 表示映射的定义域, $m[p]$ 表示 p 在 m 中的映射结果, $m \rceil p$ 表示从映射定义域中去除 p, 用 $m[p \coloneqq v]$ 表示在 m 中将 p 映射为 v. 关于列表操作, 对任意有限长的序列 l 来讲, 下文用 $|l|$ 表示序列长度, $l_1 \cdot l_2$ 表示对序列的尾部追加, update i v l 表示将 l 中的第 i 个元素更新为 v, 用 nth i l 表示返回

序列 l 中的第 i 个元素. 关于文件原子操作和块原子操作的求值规则, 分别如下文的定义 2.6 和定义 2.7 所示.

定义 2.6(文件原子操作的求值规则).

$$\frac{f \notin \mathrm{dom}(s_f)}{(\text{fcreate } lb)/(s_f,s_b) \Downarrow f/(s_f[f \coloneqq lb],s_b)} \qquad \frac{f \in \mathrm{dom}(s_f)}{(\text{fdelete } f)/(s_f,s_b) \Downarrow tt/((s_f \rceil f),s_b)}$$

$$\frac{f \in \mathrm{dom}(s_f)}{(\text{fget } f)/(s_f,s_b) \Downarrow (s_f[f])/(s_f,s_b)} \qquad \frac{f \in \mathrm{dom}(s_f)}{(\text{fsize } f)/(s_f,s_b) \Downarrow (|s_f[f]|)/(s_f,s_b)}$$

$$\frac{f \in \mathrm{dom}(s_f)}{(\text{nthblk } f\ n)/(s_f,s_b) \Downarrow (\text{nth } n\ s_f[f])/(s_f,s_b)}$$

$$\frac{f \in \mathrm{dom}(s_f)}{(\text{attach } f\ lb)/(s_f,s_b) \Downarrow tt/(s_f[f \coloneqq (s_f[f] \cdot lb)],s_b)}$$

$$\frac{f \in \mathrm{dom}(s_f)}{(\text{fset } f\ n\ b)/(s_f,s_b) \Downarrow tt/\left(s_f[f \coloneqq \text{update } n\ b\ s_f[f]],s_b\right)}$$

文件原子操作只会更新 s_f, 第 1 行说明创建文件和删除文件: 以块地址序列 lb 创建文件, 返回值为新的文件地址 f, 同时, s_f 中会将 f 映射到 lb; 以地址 f 删除文件, 返回值为 unit 类型的无意义值 tt, 同时在 s_f 的定义域中去除 f.

第 2 行说明获取文件中块序列和统计文件中块的数量: 以文件地址 f 获取文件中块序列的操作, 返回值为与文件相关联的块地址序列, 该操作不改变状态; 获取文件块数量的操作, 与读取文件块序列同理, 它的返回值是块地址序列的长度, 同样不改变系统状态.

第 3 行说明索引文件中特定块的地址: 以文件地址 f 和整数 n 来索引目标块, 返回值是文件中第 n 个块的地址. 第 4 行说明文件的追加: 在与文件地址 f 关联的块地址序列尾部, 追加新的序列 lb, 对应更新 s_f 同时返回值为 tt. 最后一行说明修改文件中目标块的地址: 用块地址 b 修改 f 索引结果中的第 n 个元素, 以此更新 s_f 并返回 tt.

定义 2.7(块原子操作的求值规则).

$$\frac{b \notin \mathrm{dom}(s_b)}{(\text{bcreate } ln)/(s_f,s_b) \Downarrow b/(s_f,s_b[b \coloneqq ln])} \qquad \frac{b \in \mathrm{dom}(s_b)}{(\text{bdelete } b)/(s_f,s_b) \Downarrow tt/(s_f,(s_b \rceil b))}$$

$$\frac{b \in \mathrm{dom}(s_b)}{(\text{bget } b)/(s_f,s_b) \Downarrow (s_b[b])/(s_f,s_b)} \qquad \frac{b \in \mathrm{dom}(s_b)}{(\text{bsize } b)/(s_f,s_b) \Downarrow (|s_b[b]|)/(s_f,s_b)}$$

$$\frac{b \in \mathrm{dom}(s_b)}{(\text{append } b\ ln)/(s_f,s_b) \Downarrow tt/\left(s_f,s_b[b \coloneqq (s_b[b] \cdot ln)]\right)}$$

块的原子操作只会更新 s_b, 第 1 行说明创建块和删除块: 以整数序列 ln 创建块, 会返回新的块地址 b, 并在 s_b 中新建 b 与 ln 的映射; 以块地址 b 删除块, 返回 tt 的同时, 会在 s_b 的定义域中去除 b. 第 2 行说明读取块的内容和统计块的大小: 以块地址 b 进行读取, 返回值为该块所存储的整数序列, 同时不改变状态; 统计块的大小, 返回值是块中整数序列的长度, 同样不改变状态. 最后一行说明块内容的追加: 在块地址映射结果的尾部追加新内容 ln, 返回值为 tt, 同时更新 s_b.

值得注意的是, 工具中构建的建模语言, 保留了实际中 HDFS 的一些设定[29]. 比如, 对于文件删除来讲, HDFS 不会直接清除文件中数据块的内容, 而是会将它们暂时保留防止误删. 其次, HDFS 的简单一致性原则, 提到了数据块的内容不允许被修改. 在后续的迭代开发中, HDFS 又引入了追加命令来提高系统的可操作性. 这些设定在工具的建模语言中都有所对应.

3 块云存储堆谓词和蕴含

本节介绍对 CBS 系统状态性质的描述以及这些描述之间的逻辑关系. 工具运用在堆上的谓词来描述状态

的性质. 一个CBS堆谓词, 描述了一个CBS堆的性质, 它由CBS堆直接定义. 同时, 按照CBS堆的定义, CBS堆谓词又能被细分为两层内部堆谓词的组合形式. 本节也相应给出了文件堆谓词和块堆谓词的定义.

相比较而言, 细化前的 CBS 堆谓词, 适用于制定 frame 规则等与语言细节无关的推理规则; 细化后的规则能够更精确地描述各项原子操作对状态性质的影响. 通过定义CBS堆之间的蕴含关系, 工具说明了CBS堆谓词间的逻辑关系. 尤其证明了 CBS 堆谓词细化前后, 分离逻辑运算的等价性. 它能够打通宏观和微观视角下对性质描述上的差异, 更能支持各种推理规则之间的相互作用, 这些对后续的推理尤为关键.

3.1 CBS堆谓词

要讨论对程序行为的描述, 首先需要用断言的形式来描述程序执行的某个时刻以及系统状态的相应性质. 断言就是一系列关于系统状态的命题. 验证工具中, 我们用CBS堆谓词的形式来描述一个CBS堆的性质.

定义 3.1(CBS 堆谓词). CBS 堆谓词的类型为“heap → Prop”.

CBS堆谓词通过CBS堆定义, 它为宏观角度下分析CBS的系统性质提供了可能. CBS堆谓词采用“$\lambda h.P$”的定义形式, 其中, P 是有关 h 的命题公式. 按照λ演算中的归约, 一个 CBS 堆谓词 H 在某一状态 h 中可以被满足, 能够被记为“$H\ h$”, 它直观地被理解为: 代入堆实例后的命题公式成立. CBS 堆谓词主要运算符的具体定义和解释如定义 3.2 所示. 其中, 有关 CBS 堆之间运算的含义, 可参照第 2.1 节中的符号约定.

定义 3.2(CBS 堆谓词的运算符).

运算符	记号	定义
空堆	[ ]	$\lambda h.h=\oslash$
纯断言	$[P]$	$\lambda h.h=\oslash\wedge P$
分离合取	$H_1\star H_2$	$\lambda h.\exists h_1h_2.(h_1\perp h_2)\wedge(h=h_1\uplus h_2)\wedge(H_1\ h_1)\wedge(H_2\ h_2)$
存在量词	$\dot{\exists}x.H$	$\lambda h.\exists x.(H\ h)$
全称量词	$\dot{\forall}x.H$	$\lambda h.\forall x.(H\ h)$

空堆[]刻画了一个空的 CBS 堆; 纯断言[P]除了刻画一个空的堆之外, 还额外表明命题 P 的成立; 分离合取 $H_1\star H_2$ 刻画了一个CBS堆可以被分割为两个不相交的堆 h_1 和 h_2, 这两个堆分别满足 H_1 和 H_2; 存在和全称量词允许在 CBS 堆谓词(heap→Prop)层面进行量化, 这与原生 Coq 在命题(Prop)层面的量化不同. 注意 $\dot{\exists}x.H$ 和 $\dot{\forall}x.H$ 可以不受限制地量化任意类型的值, 甚至允许量化堆谓词或命题.

如上文提到的, CBS 堆谓词间的运算在宏观层面, 而原子操作对系统状态的更新, 则发生在内部的存储层级, 因此我们还需要对 CBS 的状态性质进行更加细化的描述.

对应文件堆和块堆,下面我们分别定义描述其性质的文件堆谓词和块堆谓词.

定义 3.3(文件堆谓词). 文件堆谓词的类型为“heapf→Prop”.

定义 3.4(块堆谓词). 块堆谓词的类型为“heapb→Prop”.

随后, 令 H_f 表示一个文件堆谓词, H_b 表示一个块堆谓词. 一个细化版本的 CBS 堆谓词就形如$\langle H_f,H_b\rangle$, 按照类型系统, 它仍然是一个CBS堆谓词, 可它同时又能够描述内部层级状态的性质. 对于一个CBS堆h, 我们用 $h.f$ 表示其内部的文件堆, $h.b$ 表示其块堆, 分别对应二元组 (h_f,h_b) 的前件和后件. 下面定义对 CBS 堆谓词的细化.

定义 3.5(对 CBS 堆谓词的细化). 一个 CBS 堆谓词被细化后形如$\langle H_f,H_b\rangle$, 它表示一个 CBS 堆内部的文件堆能满足 H_f, 并且块堆能满足 H_b, 即 $\langle H_f,H_b\rangle\ \equiv\ \lambda h.\big((H_f\ h.f)\wedge(H_b\ h.b)\big)$.

可以观察到, 对 CBS 堆谓词的细化显然涉及到内部堆谓词的具体含义. 下面, 我们分别定义文件堆谓词和块堆谓词的核心运算符. 所谓的核心运算符是指, 同类型的堆谓词都可以运用这些运算符来表示.

文件堆谓词是在文件堆上的谓词, 它用于描述 CBS 中文件层状态的性质. 定义同样采用上述 $\lambda h_f.P$ 的形式, 区别是这里由文件堆参与归约. 下面, 令 f 表示文件地址, lb 表示块地址序列, null 为空地址. 文件堆谓词的核心运算符如定义 3.6 所示.

定义 3.6(文件堆谓词的核心运算符).

运算符	记号	定义
空文件堆	$[\,]_f$	$\lambda h_f.\, h_f = \varnothing$
文件纯断言	$[P]_f$	$\lambda h_f.\, h_f = \varnothing \wedge P$
文件单堆	$f \mapsto_f lb$	$\lambda h_f.\, h_f = (f \to lb) \wedge f \neq \text{null}$
文件分离合取	$H_f \star_f H'_f$	$\lambda h_f.\, \exists h_f^1 h_f^2.\, (h_f^1 \perp h_f^2) \wedge (h_f = h_f^1 \uplus h_f^2) \wedge (H_f\ h_f^1) \wedge (H'_f\ h_f^2)$
文件存在量词	$\exists_f x.H_f$	$\lambda h_f.\, \exists x.\, (H_f\ h_f)$
文件全称量词	$\forall_f x.H_f$	$\lambda h_f.\, \forall x.\, (H_f\ h_f)$

空文件堆 $[\,]_f$ 刻画了一个空的文件堆; 文件纯断言 $[P]_f$ 在空文件堆的基础上, 还表明了一个命题公式的成立; 文件单堆 $f \mapsto_f lb$ 刻画了一个文件堆是单例映射 $f \to lb$, 即该映射的定义域中仅有 f, 并且它的取值为 lb. 单堆还能表明文件地址不为空, 即块序列不会被存放在空的文件地址中; 文件分离合取, 表明一个文件堆可以被分割为两个不相交的堆, 它们分别使 $\star_f$ 符号左右的文件堆谓词满足; 文件堆谓词中全称和存在量词, 都是在文件堆谓词层面(heapf→Prop)的量化, 这里用下标 f 作区分.

与文件堆谓词的思想相类似, 块堆谓词核心运算符的定义采用 $\lambda h_b.\, P$ 的形式, 也就是在块堆上的谓词. 令 b 表示数据块地址, ln 表示整数序列, 块堆谓词的核心运算符如定义 3.7 所示.

定义 3.7(块堆谓词的核心运算符).

运算符	记号	定义
空块堆	$[\,]_b$	$\lambda h_b.\, h_b = \varnothing$
块纯断言	$[P]_b$	$\lambda h_b.\, h_b = \varnothing \wedge P$
块单堆	$b \mapsto_b ln$	$\lambda h_b.\, h_b = (b \to ln) \wedge b \neq \text{null}$
块分离合取	$H_b \star_b H'_b$	$\lambda h_b.\, \exists h_b^1 h_b^2.\, (h_b^1 \perp h_b^2) \wedge (h_b = h_b^1 \uplus h_b^2) \wedge (H_b\ h_b^1) \wedge (H'_b\ h_b^2)$
块存在量词	$\exists_b x.H_b$	$\lambda h_b.\, \exists x.\, (H_b\ h_b)$
块全称量词	$\forall_b x.H_b$	$\lambda h_b.\, \forall x.\, (H_b\ h_b)$

同理, 空块堆刻画了空的块堆; 块纯断言在空块堆的基础上, 额外表明一个命题的成立; 块单堆刻画了块堆是单例映射($b \to ln$), 同时表明块地址不为空; 块分离合取刻画了块堆可被分离为不相交的两个堆, 它们可以分别满足对应的块堆谓词; 块存在量词和块全称量词则是在块堆谓词层面(heapb→Prop)的量化.

3.2 蕴含关系

蕴含(entailment)可以从语义的角度表示堆谓词之间的有序关系, 它一般被用于构建推理规则和堆谓词的运算律. 本文中, 我们以"$H_1 \vdash H_2$"表示 H_1 能够蕴含 H_2.

定义 3.8(CBS 堆谓词的蕴含). 对于任意 CBS 堆谓词 H_1 和 H_2, 如果任意满足 H_1 的 CBS 堆都能使 H_2 满足, 那么称 H_1 蕴含 H_2, 记为 $H_1 \vdash H_2$, 即 $H_1 \vdash H_2 \equiv \forall h.\, (H_1\ \ h) \Rightarrow (H_2\ \ h)$.

CBS 堆谓词关于蕴含可以满足自反、传递和反对称的有序关系, 它们依次对应下面引理中的 3 个性质.

引理 3.1(蕴含在 CBS 堆谓词集上定义的有序关系).

$$\frac{}{H \vdash H} \qquad \frac{H_1 \vdash H_2 \qquad H_2 \vdash H_3}{H_1 \vdash H_3} \qquad \frac{H_1 \vdash H_2 \qquad H_2 \vdash H_1}{H_1 = H_2}$$

其中, 反对称性给出两个 CBS 堆谓词相等的判断依据, 借此我们证明了传统分离合取的运算律, 对于 CBS 堆谓词仍然成立: 分离合取 $\star$ 满足交换律和结合律, 任何与空堆[]运算的堆谓词都不会发生改变. 即运算 $\star$ 和单位元[]在 CBS 堆谓词集合上能够构成一个可交换幺半群.

特别地, 对于两个 CBS 堆谓词间的分离合取, 反对称性可以说明该运算在宏观和微观角度上的等价性. 我们证明, 两个细化后 CBS 堆谓词在宏观上的分离合取, 等价于它们内部分别进行分离合取. 这个等价性将内部状态和 CBS 的状态关联了起来, 为后续证明推理规则提供了基础, 它的形式描述如下.

引理 3.2(在宏观和微观角度中分离合取的等价性). 对于任意两个细化后的 CBS 堆谓词 $\langle H_f^1, H_b^1\rangle$ 和 $\langle H_f^2, H_b^2\rangle$, 它们之间分离合取的成立, 当且仅当由它们内部堆谓词分离合取后所组成的新谓词成立. 即

$$\langle H_f^1, H_b^1\rangle \star \langle H_f^2, H_b^2\rangle \;=\; \langle (H_f^1 \star_f H_f^2), (H_b^1 \star_b H_b^2)\rangle .$$

此外, 下面的引理支持在分离合取的运算中, 将量词和纯断言提取出来, 这对简化实际中的推理很有帮助.

引理 3.3(蕴含中量词和纯断言的性质).

$$\frac{P \Rightarrow (H \vdash H')}{([P] \star H) \vdash H'} \qquad \frac{(H \vdash H') \quad P}{H \vdash (H' \star P)} \qquad \frac{\dot\forall x.(H \vdash H')}{(\dot\exists x.H) \vdash H'} \qquad \frac{H \vdash ([a/x]H')}{H \vdash (\dot\exists x.H')}$$

蕴含关系还能表达一些特定 CBS 堆谓词的隐含性质. 例如, 由块单堆 $b \mapsto_b ln$ 能够推理出 $b \neq$null. 同理, 对于一个形如 $(b \mapsto_b ln_1) \star_b (b \mapsto_b ln_2)$ 的块分离合取, 通过对同一个块地址的重复描述能够推出矛盾, 因为块分离合取的定义中, 提到了块堆不相交. 根据相同块地址矛盾, 能够进一步得出不同文件无法共享数据块. 举反例说明, 假设两个不同文件关联了一个相同的块, 那么在描述文件的整体存储状态时, 势必需要对应地刻画内部的数据块. 但是, 如果用分离合取同时描述这两个文件, 那么根据定理 3.2, 可以在内部的块堆谓词中构建出矛盾. 与此同时, 每个块只能属于一个文件, 也符合现实中 CBS 的实际要求. 上述性质可由下例来形式化地加以表示.

例 1(内部堆谓词的隐含性质).

$$\text{块地址非空:}\quad \langle H_f, (b \mapsto_b ln)\rangle \;\vdash\; \langle H_f, (b \mapsto_b ln)\rangle \star [b \neq \text{null}]$$

$$\text{相同块地址矛盾:}\quad \langle H_f, (b \mapsto_b ln_1) \star_b (b \mapsto_b ln_2)\rangle \;\vdash\; [\text{False}]$$

$$\text{不同文件共享块矛盾:}\quad \langle (f_1 \mapsto_f b), (b \mapsto_b ln_1)\rangle \star \langle (f_2 \mapsto_f b), (b \mapsto_b ln_2)\rangle \;\vdash\; [\text{False}]$$

3.3 关于后置条件的处理

按照第 2.3 节中定义的求值规则, 指令在执行时会输出一个返回值. 因此, 对于三元组的后置条件来讲, 它在描述输出状态性质的同时, 还需要对返回值进行描述.

定义 3.9(后置条件). 后置条件的类型为 val→heap→Prop.

下面, 令 Q 表示一个三元组中的后置条件. 为了能够更简洁地定义推理规则和规范, 仿照 $H \star H'$ 和 $H \vdash H'$ 的形式, 我们针对后置条件定义了分离合取和蕴含, 它们在本文中被记为 $Q \star H$ 和 $Q \vdash Q'$, 这里, 对运算符做出的一点改变, 只是为了标记运算中额外引入了值.

定义 3.10(后置条件与 CBS 堆谓词的分离合取). 由后置条件的类型可知, 向 Q 代入一个返回值 v 后, $(Q\ v)$ 的类型为 CBS 堆谓词. 关于后置条件的分离合取, 可以直接沿用 CBS 堆谓词分离合取的定义. 相关的形式化定义为 $Q \star H \equiv \lambda v.((Q\ v) \star H)$.

后置条件间的蕴含关系, 同样是 CBS 堆谓词间蕴含的推广.

定义 3.11(后置条件间的蕴含). 对于后置断言来讲, Q 蕴含 Q'当且仅当对于任意的值 v, CBS 堆谓词$(Q\ v)$蕴含$(Q'\ v)$. 符号表示为 $Q \vdash Q' \equiv \dot\forall v.((Q\ v) \vdash (Q'\ v))$.

4 三元组和推理规则

本节介绍工具中如何描述程序的行为以及为验证程序而制定的一系列推理规则. 工具定义的CBS分离逻辑三元组, 能够描述程序执行前后 CBS 系统的性质. 基于分离逻辑的思想, 该三元组只需描述与程序执行有关的系统状态. 针对工具中所编写的程序, 用于描述其行为的三元组被称为这个程序的规范. 随后, 工具为每个新引入的原子操作, 都制定了相应的推理规则. 同时, 我们还说明了之前工具中的一些推理规则, 尤其是 frame 规则, 仍能被应用于有关 CBS 系统的推理中.

4.1 CBS分离逻辑三元组

之前工具中的分离逻辑三元组, 是通过 Hoare 三元组定义的[27]. 根据其定义思想, 一个 Hoare 三元组用于描述程序在全局状态中的执行, 而分离逻辑三元组只需描述与程序执行有关的部分系统状态, 这对应了分离逻辑小规模规范的特性.

按照这样的定义方式, 我们如果要定义针对 CBS 堆谓词的分离逻辑三元组, 就要先重新定义 Hoare 三元组. 同样在本工具中, Hoare 三元组描述程序执行时的全局 CBS 状态, 分离逻辑三元组只需描述与程序执行相关的部分 CBS 状态. 也就是说, Hoare 三元组是全局推导, 而分离逻辑三元组是关于 CBS 的局部推导.

本文中, 一个描述某个程序行为的 Hoare 三元组, 被写为“$\overset{HOARE}{}\{H\}\,t\,\{Q\}$”. 它意味着对于任意的全局 CBS 状态 s 来讲, 如果 s 能够满足前置条件 H, 并且在 s 下执行 t 后, 程序终止于全局状态 s', 同时返回值为 v, 那么 s'和 v 可以满足后置条件 Q. 可见, Hoare 三元组的成立与 t 的终止有关, 因此它断言了程序 t 的完全正确性.

定义 4.1(Hoare 三元组的完全正确性).

$$\overset{HOARE}{}\{H\}\,t\,\{Q\} \equiv \dot{\forall} s.\ (H\ \ s) \Rightarrow \exists v.\exists s'.(t/s \Downarrow v/s') \wedge (Q\ \ v\ \ s').$$

相比于 Hoare 三元组对程序在全局状态下的描述, 分离逻辑三元组只需描述程序在部分状态下的行为, 而无需描述不涉及程序执行的无关状态. 通过对这些“无关状态”进行全称量化, 可以直接将这两种形式的三元组关联起来.

本文中, 一个描述某程序行为的 CBS 分离逻辑三元组, 形如“$\{H\}\,t\,\{Q\}$”, 它断言对于任意描述了无关状态的 CBS 堆谓词 H', Hoare 三元组 $\dot{\cdot}^{HOARE}\{H \star H'\}t\{Q \star H'\}$. 都成立. 这样的三元组被称为关于 t 的规范.

定义 4.2(CBS 分离逻辑三元组的完全正确性).

$$\{H\}t\{Q\} \equiv \dot{\forall} H' \cdot^{HOARE} \{H \star H'\}t\{Q \star H'\}.$$

考虑到原子操作会更新内部状态, 我们展开了上述定义中的 CBS 堆谓词, 从而可以得到一个细化版的 CBS 分离逻辑三元组. 注意, 展开只是在语法形式上的变换, 展开前后三元组的类型一致. 关于细化版 CBS 分离逻辑三元组的验证, 可以相应地通过将 s 展开为(s_f,s_b), 直接运用语义进行证明.

定义 4.3(细化版的 CBS 分离逻辑三元组).

$$\left\{\langle H_f,H_b\rangle\right\}t\left\{\lambda v.\langle H_f',H_b'\rangle\right\} \equiv \dot{\forall} H_f^0 H_b^0.\ ^{HOARE}\left\{\langle H_f,H_b\rangle \star \langle H_f^0,H_b^0\rangle\right\}\ t\ \left\{(\lambda v.\langle H_f',H_b'\rangle) \star \langle H_f^0,H_b^0\rangle\right\}.$$

由上文的引理 3.2 可知, 宏观上的分离合取与内部分别作分离合取等价, 借此我们能够规范每个原子操作在其内部层级的行为, 随后运用推理规则和蕴含关系, 就可以将内部或局部的验证结果推广到宏观层面以及全局的 CBS 上, 从而实现对管理程序的正确性验证.

4.2 推理规则

工具中的推理规则主要分为 3 个方面: 结构化规则、项的推理规则、原子操作的规范. 结构化规则与语言细节无关; 项的推理规则只针对项的构造子, 与原子操作无关; 原子操作的规范, 是为每个新引入的原子操作所制定的推理规则. 以 CBS 分离逻辑三元组的形式, 对于之前工具中前两类的推理规则, 我们进行了重新定义和证明. 尤为关键的是, 我们为新引入的原子操作制定了一系列的推理规则. 下面分别对这些推理规则进行介绍.

结构化规则包含顺序复合语句(consequence)规则、frame 规则以及关于纯断言和存在量词的提取规则.

引理 4.1(结构化规则).

$$\frac{H \vdash H' \quad \{H'\}t\{Q'\} \quad Q' \vdash Q}{\{H\}t\{Q\}} \qquad \frac{\{H\}t\{Q\}}{\{H \star H'\}t\{Q \star H'\}} \qquad \frac{P \Rightarrow \{H\}t\{Q\}}{\{[P] \star H\}t\{Q\}} \qquad \frac{\dot{\forall} x.\{H\}t\{Q\}}{\left\{\dot{\exists} x.H\right\}t\{Q\}}$$

顺序复合语句规则支持强化前置条件和弱化后置条件. Frame 规则断言如果项 t 在给定的部分状态下能够正确执行, 那么 t 也可以在更大部分的状态下正确执行. 随后的两个提取规则, 支持将纯断言和量词从前置条

件中提取出来, 作用到整个三元组上. 在工具中, 两个提取规则同样适用于细化后的三元组, 前置条件中的内部堆谓词带有的存在量词或纯断言, 也可以被提取作用到整个三元组. 下面的例子, 就是与文件堆谓词相关的提取规则.

例 2(对文件堆谓词中存在量词和纯断言的提取规则).

$$\frac{P \Rightarrow \{\langle H_f, H_b\rangle\}\ t\ \{Q\}}{\{\langle([P]_f \star_f H_f), H_b\rangle\}\ t\ \{Q\}} \qquad \frac{\dot{\forall} x.\{\langle H_f, H_b\rangle\}\ t\ \{Q\}}{\{\langle(\exists_f x.H_f), H_b\rangle\}\ t\ \{Q\}}$$

此外, 考虑到Coq中对于命题的证明往往是自底向上的[21], 而实际验证中, 待证命题往往是形如$\{H\}t\{Q\}$的三元组, 这就意味着很难直接调用frame规则. 一般地, 人们通常需要先运用consequence规则, 将待证命题转换为$\{H \star H'\}t\{Q \star H'\}$的形式, 才能继续调用 frame 规则. 工具中引入的推论-框架(consequence-frame)规则, 通过整合这两个规则直接达到同样的效果, 该规则可以直接证明形如$\{H\}t\{Q\}$的三元组, 而不受前置和后置条件的限制.

引理 4.2(推论-框架规则).

$$\frac{H \vdash H_1 \star H_2 \quad \{H_1\}\ t\ \{Q_1\} \quad Q_1 \star H_2 \vdash Q}{\{H\}\ t\ \{Q\}}$$

项的推理规则包含了每一个项构造子对应的推理规则, 它们与原子操作无关, 如下所示.

引理 4.3(项的推理规则).

$$\frac{H \vdash (Q\ v)}{\{H\}\ v\ \{Q\}} \qquad \frac{H \vdash (Q\ (\hat{\lambda} x.t))}{\{H\}\ (\lambda x.t)\ \{Q\}} \qquad \frac{H \vdash \left(Q\ (\hat{\mu} F.\lambda x.t)\right)}{\{H\}\ (\mu F.\lambda x.t)\ \{Q\}}$$

$$\frac{\{H\}\ ([v_1/x]t)\ \{Q\}}{\{H\}\ ((\hat{\lambda} x.t)\ v_1)\ \{Q\}} \qquad \frac{v_1 = \hat{\mu} F.\lambda x.t \quad \{H\}\ ([v_2/x][v_1/F]t)\ \{Q\}}{\{H\}\ (v_1\ v_2)\ \{Q\}}$$

$$\frac{\{H\}\ t_1\ \{\lambda v.H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1; t_2)\ \{Q\}} \qquad \frac{\{H\}\ t_1\ \{Q'\} \qquad \dot{\forall} v.\{Q'\ v\}\ ([v/x]t_2)\{Q\}}{\{H\}\ (\text{let}\ \ x = t_1\ \text{in}\ \ t_2)\ \{Q\}}$$

$$\frac{(be = \text{true}) \Rightarrow \{H\}\ t_1\ \{Q\} \qquad (be = \text{false}) \Rightarrow \{H\}\ t_2\ \{Q\}}{\{H\}\ (\text{if}\ be\ \text{then}\ t_1\ \text{else}\ t_2)\ \{Q\}}$$

第 1 行的规则适用于对值封闭的项: 值、非递归和递归程序的求值是它们本身, 并不会改变系统状态. 第 2 行的规则适用于验证程序的调用, 在证明引入特定参数后来调用程序的规范时, 需要首先运用相应的规则将参数代入到程序主体. 第3行的规则适用于验证顺序语句, 两种顺序语句的不同之处在于, let绑定会将中间结果代入到随后执行的语句中. 第 4 行的规则适用于条件语句的验证.

上述两类规则, 都只是关于之前推理规则在本工具中的移植. 下面我们将重点介绍原子操作的推理规则.

原子操作的规范分为文件和块原子操作的规范, 它们是不带有任何前提的、关于文件和块原子操作的推理规则. 首先是文件原子操作的规范.

引理 4.4(文件原子操作的规范).

$$\{\langle[\text{nodup}(lb)]_f, H_b\rangle\} \quad (\text{fcreate}\ \ lb) \quad \{\lambda r.\langle(\exists_f f.[r = f]_f \star_f (f \mapsto_f lb)), H_b\rangle\}$$

$$\{\langle(f \mapsto_f lb), H_b\rangle\} \quad (\text{fdelete}\ \ f) \quad \{\lambda \_.\langle([\ ]_f), H_b\rangle\}$$

$$\{\langle(f \mapsto_f lb), H_b\rangle\} \quad (\text{fget}\ \ f) \quad \{\lambda r.\langle([r = lb]_f \star_f (f \mapsto_f lb)), H_b\rangle\}$$

$$\{\langle(f \mapsto_f lb), H_b\rangle\} \quad (\text{fsize}\ \ f) \quad \{\lambda r.\langle([r = |lb|]_f \star_f (f \mapsto_f lb)), H_b\rangle\}$$

$$\{\langle(f \mapsto_f lb), H_b\rangle\} \quad (\text{nthblk}\ \ f\ \ n) \quad \{\lambda r.\langle([r = (\text{nth}\ n\ lb)]_f \star_f (f \mapsto_f lb)), H_b\rangle\}$$

$$\{\langle[\text{nodup}(lb')]_f \star_f f \mapsto_f lb, H_b\rangle\} \quad (\text{attach}\ \ f\ \ lb') \quad \{\lambda \_.\langle(f \mapsto_f (lb \cdot lb')), H_b\rangle\}$$

$$\{\langle(f \mapsto_f lb), H_b\rangle\} \quad (\text{fset}\ \ f\ \ n\ \ b) \quad \{\lambda \_.\langle(f \mapsto_f \text{update}\ \ n\ \ b\ \ lb), H_b\rangle\}$$

创建文件 fcreate *lb* 可以在空的文件堆上执行, 同时 *lb* 中不能有重复的块地址, 这在规范中由文件纯断言 $[\text{nodup}(lb)]_f$ 来描述. 其中, nodup(*l*) 是有关列表 *l* 的命题, 它断言 *l* 中不存在重复的元素. 这个操作结束后, 文件层中新引入一个文件单堆 $f \mapsto_f lb$, 这个单堆位于某个文件地址 *f* 上, 内容为块地址序列 *lb*. 同时, 操作在执行结束后, 会输出这个新文件地址 *f*, 我们用 $[r{=}f]_f$ 来描述程序执行预期的返回值.

删除文件 fdelete *f* 需要前置条件中提到待操作单元的存储情况, 这个存储单元由 $f \mapsto_f lb$ 描述. 三元组的后置条件断言文件堆为空, 反映了操作对存储单元的释放. 这个规范同时还保证了文件 *f* 被删除后, 不能再对 *f* 执行任何操作. 注意, 此时文件关联的数据块的内容并没有被直接删除, 这是考虑到 HDFS 中不会立即释放数据块内容的设定. 如果想要彻底清除文件内容, 可以通过由删除块和删除文件组成的复合语句来表示.

对于文件 3 种不同的读取方式, 它们的规范大致同理, 并且也都不会改变系统状态. 读取文件 fget *f* 需要一个被描述为 $f \mapsto_f lb$ 的文件存储单元作为前置条件, 该操作执行的返回值是对应的块地址序列 *lb*. 获取文件块的数量 fsize *f*, 需要同样的前置条件, 来表明存在一个待读取的文件存储单元. 这个操作执行的返回值是块地址序列的长度. 索引第 *n* 个块 nthblk *f n*, 也需要一个文件单元的存在, 它的返回值是块序列中的第 *n* 个块的地址.

最后, 针对文件两种不同的修改方式, 它们的规范也大致同理, 其返回值都可以被忽略. 文件尾部追加 attach *f lb*′, 需要一个可被操作的存储单元 $f{\mapsto_f}\ lb$, 同时还需要表明新的块序列 *lb*′中没有重复地址. 这个操作会将 *f* 关联的块序列更新为 *lb*·*lb*′, 也就是两个块序列的连接. 修改第 *n* 个块地址 fset *f n b*, 也需要存在被操作的文件单元, 这个操作将 *f* 中的第 *n* 个块更新为新的块地址, 也就是将 *f* 的映射结果更新为 update *n b lb*.

引理 4.5(块原子操作的规范).

$$\{\langle H_f, [\ ]_b\rangle\} \quad (\text{bcreate } ln) \quad \{\lambda r.\langle H_f, (\exists_b b.[r=b]_b \star_b (b \mapsto_b ln))\rangle\}$$

$$\{\langle H_f, (b \mapsto_b ln)\rangle\} \quad (\text{bdelete } b) \quad \{\lambda \_.\langle H_f, ([\ ]_b)\rangle\}$$

$$\{\langle H_f, (b \mapsto_b ln)\rangle\} \quad (\text{bget } b) \quad \{\lambda r.\langle H_f, ([r=ln]_b \star_b (b \mapsto_b ln))\rangle\}$$

$$\{\langle H_f, (b \mapsto_b ln)\rangle\} \quad (\text{bsize } b) \quad \{\lambda r.\langle H_f, ([r=|ln|]_b \star_b (b \mapsto_b ln))\rangle\}$$

$$\{\langle H_f, (b \mapsto_b ln)\rangle\} \quad (\text{append } b\ ln') \quad \{\lambda \_.\langle H_f, (b \mapsto_b (ln \cdot ln'))\rangle\}$$

创建块 bcreate *ln* 可以在空的块堆上执行, 它会向块层的状态中引入一个新的块单堆 $b{\mapsto_b}\ ln$, 同时返回值为新建块的地址 *b*. 删除块 bdelete *b* 需要存在一个可操作的块单元, 规范的后置条件断言了块堆为空, 并且返回了一个可以被忽略的值. 这个规范描述了块存储单元的释放, 同时保证了被删除的块不能再被做任何的操作. 读取块内容 bget *b* 和获取块大小 bsize *b* 都需要一个可读取的块单元, 它们不会修改块堆的状态, 并且分别会返回块内容 *ln* 和块大小|*ln*|. 最后, 块内容追加 append *b ln*′需要存在被操作的块, 它在块堆中将 *b* 对应的值更新为(*ln*·*ln*′), 也就是两段块内容的连接.

需要注意的是, 块删除的规则只单纯地表明了对块的操作, 没有刻画对文件有效性的影响. 这是因为, 按照实际 CBS 的协议, 用户无法直接删除指定的数据块, 只有在回收失效文件的空间时, 块删除才会由管理程序调用. 也就是说, 从环节来讲, 文件的失效发生在块删除之前. 此外, 在描述文件整体的存储状态时, 我们需要刻画其每个数据块的内容. 对于一个已删除的块, 当通过块单堆的形式描述它时, 由于此时的块地址为空, 会使得整体的 CBS 堆谓词为假, 由此, 本文模型能够体现被删除块对于文件的影响.

5 验证程序实例

本节介绍一些关键的代码表示, 以及展示工具对实际程序的编码和验证过程. 通过介绍一些关键的代码表示, 帮助读者理解验证实例, 同时有关 frame 规则的实现和证明, 说明了工具具备局部推导的能力. 本节引入 3 个验证实例: 对“移动数据块”操作的验证, 说明工具能够推理与文件和块有关的大部分存储操作; 对“读取文件内容”的验证, 说明工具支持验证递归程序; 对“清除文件内容”的验证, 说明工具具有一定的自动化推理能力.

5.1 代码表示

截止到目前, 本文涉及的诸多表示方法, 大多是为了方便读者理解而选取的数学符号, 它们都对应了工具实现中的实际 Coq 代码. 下面, 为了说明工具对程序实例的验证过程, 有必要先简单引入文中数学符号对应的 Coq 代码表示, 并介绍一些关键的实现细节(见表 2).

表 2 工具中对各种堆谓词的代码表示

CBS 堆谓词的代码记号							
运算符	空堆	纯断言	分离合取	存在量词	全称量词	后置条件	细化的堆谓词
数学符号	[ ]	$[P]$	$H_1 \star H_2$	$\dot{\exists} x.H$	$\dot{\forall} x.H$	$\lambda x.\, H$	$\langle H_f, H_b \rangle$
代码表示	\[ ]	\[P]	H_1* H_2	\exists x, H	\forall x, H	fun x=>H	\R[H_f,H_b]
文件堆谓词的代码记号							
运算符	空文件堆	文件纯断言	文件单堆	文件分离合取	文件存在量词	文件全称量词	
数学符号	$[\,]_f$	$[P]_f$	$f \mapsto_f lb$	$H_f \star_f H_f'$	$\exists_f x.H_f$	$\forall_f x.H_f$	
代码表示	\f[ ]	\f[P]	f~f~>lb	H_f \f* H_f'	\exists f x, H_f	\forallf x, H_f	
块堆谓词的代码记号							
运算符	空块堆	块纯断言	块单堆	块分离合取	块存在量词	块全称量词	
数学符号	$[\,]_b$	$[P]_b$	$b \mapsto_b ln$	$H_b \star_b H_b'$	$\exists_b x.H_b$	$\forall_b x.H_b$	
代码表示	\b[ ]	\b[P]	b~b~>ln	H_b \b* H_b'	\exists b x, H_b	\forall b x, H_b	

运用上述 Coq 代码表示, 以引理 3.2 为例, 在工具中它被编码为如下引理.

```
Lemma hstar_sep: forall Hf Hb Hf' Hb',
    (\R[Hf,Hb]) \* (\R[Hf',Hb′])=\R[(Hf \f* Hf'),(Hb \b* Hb′)].
```

此外, 关于程序行为的描述, 同样我们在工具中也是先定义 CBS 的 Hoare 三元组, 再借此定义 CBS 分离逻辑三元组. 首先, 工具将 Hoare 三元组实现如下.

```
Definition hoare (t: trm)(H: hprop)(Q: val->hprop): Prop: =
  forall sf sb, H(sf,sb)->exists sf′ sb′ v, eval(sf,sb) t (sf′,sb′) v/ Q v(sf′,sb′),
```

其中, hprop 是 CBS 堆谓词类型, “eval s t s′ v”是关于项 t 的函数, 它的输入是起始状态 s 和项 t, 输出是终止状态 s'和返回值 v. 通过 eval 函数, 我们归纳定义了项 t 的求值规则, 也就是带返回值的操作语义. 工具中一个关于程序 t 的 Hoare 三元组, 就为形如“hoare t H Q”的命题.

随后, 基于 Hoare 三元组, 在工具中定义的 CBS 分离逻辑三元组如下所示.

```
Definition triple (t: trm)(H: hprop)(Q: val->hprop): Prop: =
  forall Hf Hb, hoare t(H \* \R[Hf,Hb])(Q \*+ \R[Hf,Hb]).
```

其中, Q *+ H 是 $Q \star H$ 的代码表示. 同理, 工具中对于程序 t 的分离逻辑三元组, 就为形如 triple t H Q 的命题.

如前文所说, 本文提到的所有引理和推理规则, 在工具中都有相应的实现和证明. 以分离逻辑中最为代表性的 frame 规则为例, 它的实现和证明就如下所示. 对于它的证明思想主要包括: 展开三元组的定义, 运用交换律和结合律, 运用假设前提中堆谓词的全称量化等.

```
Lemma triple_frame: forall (t: trm)(H: hprop)(Q: val->hprop)(Hf: hfprop)(Hb: hbprop),
  triple t H Q->triple t(H \* (\R[Hf,Hb]))(Q \*+ (\R[Hf,Hb])).
Proof.
  unfold triple, hoare.introv M.intros Hf' Hb′ sf sb. (*展开定义,命名前提和变量*)
  specializes M(Hf \f* Hf')(Hb \b* Hb′). (*在前提中实例化结合后的形式*)
  introv N.rewrite hstar_assoc, hstar_sep in N. (*命名和处理三元组的前置条件*)
  apply M in N as (sf'&sb′&v&N1&N2). (*获得证明后置条件必需的前提和实例变量*)
  exists sf' sb′ v.rewrite hstar_assoc, hstar_sep.splits~. (*实例化存在量词, 分别证明内部堆谓词*)
Qed.
```

在分离逻辑中, frame 规则是进行局部推导的核心. 工具对 frame 规则的证明, 说明我们在实际的推理中,

只需关注于程序执行相关的 CBS 状态, 以精确地描述和验证程序的行为, 而无需再关心不涉及程序执行的状态.

5.2 验证实例——Move

在 HDFS 中, DataNode 将块内容存放在本地磁盘, 然而节点中各磁盘之间的数据可能分布不够均匀, 这会导致频繁使用网络带宽, 降低了存储效率. 为了平衡节点内部磁盘的存储容量, HDFS 提供了一个名为 Disk Balancer 的命令行工具[35]. 该命令行会操作给定的节点, 通过在各个磁盘间移动数据块来平衡存储空间. 这个操作同时会更新 NameNode 中的元数据. 很明显, 移动数据块是平衡磁盘空间的核心操作, 它的正确性更是整个命令正确执行的根本. 因此, 有必要确保移动块的过程中不会发生存储错误, 比如块丢失或内容改变等. 对一个文件中第 n 个块的移动操作, 大体上可以被拆分为 4 个环节: 获取目标块地址、复制目标块的内容、更新文件中对应的块地址、释放目标块.

我们将移动块的过程命名为“Move_blk”, 对应上述操作环节的拆分, 将该操作在工具中编码表示如下.

```
Definition Move_blk: =
  Fun f i : =
    Let bk: =nth_blk f i in (*索引 f 中第 i 个块的地址*)
    Let bk1: =Copy_blk bk in (*复制目标块的内容, 获取新块的地址*)
    set_nth_blk f i As bk1; (*将文件中第 i 个块替换为新的块*)
    bdelete bk. (*删除掉旧的块*)
```

其中, “Fun f i”表明这个非递归程序需要两个参数, 也就是文件的地址和索引的位置; “Copy_blk”是在第 2 节提到的块复制操作; “set_nth_blk f i As bk1”对应文件修改 fset, 表示将文件 f 中第 i 个块的地址更新为 bk1.

随后, 可以用类似“Move_blk f n”的调用方式来表示对文件 f 中第 n 个块的移动操作. 关于该操作执行的正确性, 我们运用 CBS 分离逻辑三元组, 以定理的形式将其规范如下.

```
Lemma triple_Move_blk: forall (fp: floc)(bp: bloc)(lb: list bloc)(ln: list int)(n: int),
  bp=(nth n lb)-> (*bp 是目标块的地址*)
  triple(Move_blk fp n)
    (\R[fp~f~>lb,bp~b~>ln])
    (fun_=>\exists bp′, (\R[(fp~f~>(update n bp′ lb)),(bp′~b~>ln)])).
```

注意到, 三元组具有前提 bp=(nth n lb), 它假设 bp 是目标块的块地址. 随后, 三元组的含义为: 在前置条件所描述的状态中, 文件层只存储了一个文件, 该文件在地址 fp 中通过块地址序列 lb, 记录着一系列属于它的数据块. 同时, 在块层地址为 bp 的存储单元中, 确实存储着内容为 ln 的数据块. 那么程序执行结束后, 后置条件就描述了终止状态中, 一个新的数据块会被创建, 它的地址为 bp′而且内容仍为 ln. 同时, 旧的目标块会被释放, 新数据块的地址会被更新在文件相应的索引位置上.

需要注意的是, 移动块操作只与目标块和关联的文件有关, 因此在三元组的前置条件中也只描述了相应的系统状态. 同时, 根据上文我们对复制块 Copy_blk 规范的证明, 提到了新的块地址和目标块地址注定不相等. 此外, 我们对三元组所引入的前提假设, 只是为了能够聚焦于讨论各种存储操作. 事实上, 在编码中加入条件语句, 在证明中对于目标块是否存在展开归纳证明, 就能够不带前提地直接验证移动块操作.

```
Proof.
  introv->.applys* triple_app_fun2.simpls.                (*初始化证明*)
  applys triple_let triple_fget_nth_blk.ext.              (*证明索引操作, ext 能够自动提取返回值*)
  applys triple_let triple_copy_blk.ext.intros bp′.ext.   (*证明复制操作, 将新块的地址命名为 bp′*)
  applys triple_seq.                                      (*分解顺序语句*)
  apply triple_fset_nth_blk.                              (*证明更新文件的操作*)
  applys triple_conseq_frame triple_bdelete.              (*引入新的前后置条件, 证明块删除操作*)
```

```
    rewrite hstar_sep, hfstar_hempty_r, hbstar_comm.applys himpl_refl.   (*证明前置条件的蕴含*)
    intros r.rewrite hstar_sep, hfstar_hempty_r, hbstar_hempty_l.        (*改写后置条件*)
    intros h(MA&MB).
    exists bp′.splits~.   (*实例化新的块地址 bp′后, 后置条件间的蕴含成立*)
  Qed.
```

5.3 验证实例——ReadFile

读取文件的内容, 是文件存储系统中最常用的操作. 我们运用一个递归程序 Read_f, 来表示对任意大小文件的读取操作. 注意, 该程序只需输入文件地址和索引位置, 它在执行中能够自行获取文件中块的数量, 以此判断何时需要终止递归, 这样的执行方式符合实际的 CBS 操作. 该程序主要分为 3 个环节: 获取文件下数据块的数量、递归地读取每个块的内容、合并所有块的内容.

```
  Definition Read_f: =
    Fix F f i: =                    (*声明递归程序的名称: F*)
      Let n: =fsize f in            (*获取文件下块的数量*)
      Let be: =(i=n) in             (*判断此次执行的目标块是否为最后一个*)
      If_ be                        (*条件分支语句*)
      Then nil                      (*如果是, 则返回空列表*)
      Else                          (*如果不是, 则执行以下操作*)
        Let bk: =nth_blk f i in     (*索引目标块地址*)
        Let i: =i+1 in              (*计数器加 1*)
        Let ln: =bget bk in         (*读取目标块内容*)
        Let ln1: =F f i in          (*递归调用 F*)
        ln ++ ln1.                  (*合并所有的读取结果*)
```

其中, “Fix F f i”表示一个带有两个参数的递归程序, F 是它在内部递归调用时的名称, 如倒数第 2 行的“F f i”. 调用该程序需要两个参数: 文件地址和索引位置. 通过引入不同的索引位置, 该程序能够从特定的数据块开始读取文件内容, 比如读取文件 f 的全部内容, 调用方式就形如“Read_f f 0”.

需要说明的是, 在验证同时涉及了文件和块操作的程序时, 用户需要刻画目标文件的整体存储情况. 也就是在三元组的前置条件中, 明确地刻画出与文件相关联的块序列. 因为如果块的地址不明确, 那么自然就无法操作相应的目标块. 为了满足实际验证的需要, 我们可以编写以任意文件作为输入的程序来表示涉及文件和块的复杂操作, 就如 Read_f 一样. 但不建议在相应的三元组中, 量化与文件关联的块序列, 因为这会导致块地址信息的丢失. 事实上, 在程序执行前每个文件的整体存储情况都是确定的, 我们可以编写对应的前置条件.

下面就考虑这样的一种存储情况: 一个文件下有 3 个块, 每个块都正常存储着相应的块内容. 那么从头开始读取该文件, 会返回由这 3 个块组成的全部内容. 下面的规范, 就描述了对这个文件的读取操作. 为了方便理解, 我们写明了各个块的块内容, 事实上它们也可以是任意的整数序列.

```
  Lemma triple_Read_f: forall(Hf: hfprop)(Hb: hbprop)(f: floc)(bp1 bp2 bp3: bloc)(n1 n2 n3 n4 n5: int),
    Hf=(f~f~>(bp1:: bp2:: bp3:: nil))-> (*文件堆谓词: 一个文件下有 3 个块*)
    (*块堆谓词: 3 个块对应的块内容*)
    Hb=((bp1~b~>(n1:: n2:: nil)) \b* (bp2~b~>(n3:: n4:: nil)) \b* (bp3~b~>(n5:: nil)))->
    triple(Read_f f 0)
      (\R[Hf,Hb])
      (fun r=>\[r=(n1:: n2:: n3:: n4:: n5:: nil)] \* \R[Hf,Hb]).
```

这个规范断言了操作的返回值为一个由文件全部内容组成的整数序列, 并且不会对系统状态做出改变.

下面是在工具中有关它的证明脚本. 考虑到递归操作有重复执行的部分, 工具还编写了一些自动化证明脚本以缩短证明过程.

```
Proof.
  (*第 1 次递归*)
  fix_read. (*应用推理脚本, 弱化前置条件, 强化后置条件*)
  rewrite hstar_sep, hfstar_hempty_r.apply himpl_refl. (*前置条件间的蕴含成立*)
  intros r.rewrite hstar_sep, hfstar_hempty_r, hbstar_assoc.apply himpl_refl. (*后置条件间的蕴含成立*)
  ext.applys triple_let. (*提取返回值, 并进入下一次递归*)
  (*第 2 次递归*)
  fix_read.
  rewrite hstar_sep, hfstar_hempty_r, hbstar_comm, hbstar_assoc.apply himpl_refl.
  intros r.rewrite hstar_sep, hfstar_hempty_r, hbstar_assoc.apply himpl_refl.
  ext.applys triple_let.
  (*第 3 次递归*)
  fix_read.
  rewrite hstar_sep, hfstar_hempty_r, hbstar_comm, hbstar_assoc.apply himpl_refl.
  intros r.rewrite hstar_sep, hfstar_hempty_r, hbstar_assoc.apply himpl_refl.
  ext.applys triple_let.
  (*退出递归*)
  exp_fix.
  applys triple_val′.ext. (*证明递归的退出, 提取结果 nil*)
  applys triple_list_app.ext. (*证明列表追加, 提取结果(n5:: nil)*)
  applys triple_list_app.ext. (*证明列表追加, 提取结果(n3:: n4:: n5:: nil)*)
  applys triple_conseq triple_list_app. (*改变前后置条件, 证明列表追加*)
  apply himpl_refl. (*系统状态没有更新, 前置条件可以直接蕴含*)
  intros r.rew_list. (*整理列表的格式, 最终结果为(n1:: n2:: n3:: n4:: n5:: nil)*)
  rewrite hbstar_comm, hbstar_assoc.apply himpl_refl. (*后置条件蕴含成立*)
Qed.
```

5.4 验证实例——RemoveFile

在实际的 HDFS 中, 对文件的彻底清除涉及到解除文件与块之间的关联, 以及释放每个块的存储空间这两个环节. 工具中, 我们编写了一个尾递归的程序 Remove_f, 来表示对任意文件的删除操作. 该程序同样只需输入文件地址和索引位置, 就能自动判断何时终止递归. 程序主体可分为 3 个环节: 获取文件下块的数量、递归释放每个块的内容、解除文件与块的关联.

```
Definition Remove_f: =
  Fix F f i: =                    (*声明递归程序的名称: F*)
    Let n: =fsize f in            (*获取文件下块的数量*)
    Let be: =(i=n) in             (*判断此次执行的目标块是否为最后一个*)
    If_be                         (*条件分支语句*)
    Then (fdelete f)              (*如果是, 则解除文件与块的关联*)
    Else                          (*如果不是, 则执行以下操作*)
      Let bk: =nth_blk f i in     (*索引目标块地址*)
      Let i: =i+1 in              (*计数器加 1*)
```

```
bdelete bk;                    (*释放目标块的内容*)
F f i.                         (*递归调用 F*)
```

同样地, “Fix F f i”表示一个带有两个参数的递归程序, 它们分别为文件地址和索引位置. 通过形如“Remove_f f 0”的方式调用程序, 就可以表示对一个文件 f 内容的全部释放. 如上文所述, 在验证文件和块的复合操作时, 我们需要明确目标文件的整体存储情况. 那么, 接下来考虑这样的一个文件 f, 它内部共有 4 个数据块, 每个块的内容直接用任意的整数序列表示. 下面的规范, 就描述了对这个文件的整体清除操作.

```
Lemma triple_Remove_f: forall(Hf: hfprop)(Hb: hbprop)(f: floc)(bp1 bp2 bp3 bp4: bloc)(l1 l2 l3 l4: listint),
  Hf=(f~f~>(bp1:: bp2:: bp3:: bp4:: nil))-> (*文件堆谓词: 一个文件下有 4 个块*)
  (*块堆谓词: 4 个块的内容为任意的整数序列*)
  Hb=((bp1~b~>l1) \b* (bp2~b~>l2) \b* (bp3~b~>l3) \b* (bp4~b~>l4))->
  triple(Remove_f f 0)
    (\R[Hf,Hb])
    (fun_=>\R[\f[], \b[]]).
```

规范的后置断言刻画了在删除操作执行后, 文件内部的各个数据块均会被释放, 并且文件与块的关联也被解除, 同时, 程序的返回值可以被忽略. 下面是在工具中有关该规范的证明脚本. 可见运用自动化证明脚本, 我们能够相对简洁地完成对递归程序 Remove_f 的证明.

```
Proof.
  fix_rem.inner_femp.outer_emp.     (*第 1 次递归*)
  fix_rem.inner_femp.outer_emp.     (*第 2 次递归*)
  fix_rem.inner_femp.outer_emp.     (*第 3 次递归*)
  fix_rem.outer_emp.outer_emp.      (*第 4 次递归*)
  exp_fix.applys triple_fdelete.    (*终止递归*)
Qed.
```

6 相关工作

在本文中, 我们基于分离逻辑, 新提出了一种用于验证 CBS 管理程序正确性的工具. 该工具能够编码表示管理程序, 支持运用分离逻辑三元组精确地描述程序的行为, 并可以运用局部推导的特性, 将局部的性质推广到全局的 CBS 状态, 以达到对管理程序的正确性验证. 我们将相关工作分为 3 类. 首先, 讨论定理证明技术在传统存储系统中的应用; 其次, 讨论定理证明在云存储系统中的应用; 最后, 讨论基于分离逻辑开发的一些优秀的验证工具. 这些工作对于本文的研究内容都具有一定的指导意义.

形式化验证技术主要分为模型检测和定理证明. 模型检测以穷尽搜索状态空间的方式, 判断系统模型是否满足某种性质[36]. 由于容易受状态空间爆炸等问题的限制, 并且存储系统普遍又比较复杂, 模型检测一般更适用于精确刻画存储中的一些场景和问题, 如死锁、数据保密等. 相对来讲, 定理证明是基于数理逻辑的形式化验证技术, 通过抽象描述计算机程序以及大型系统, 人们可以建模出具有推导能力的公理化系统. 在随后的验证过程中, 期望的系统性质被表示成数学定理, 一般为逻辑公式. 随后, 可以通过判定该定理在公理化系统中是否可证, 从而判断系统性质能否被满足. 定理证明对于分析传统存储系统已经具备了一些成熟的成果. 针对 Unix 风格的文件系统, Arkoudas 等人提出过一个基本的形式化建模[37]. 他们将存储系统定义为文件到数据序列的映射, 这个数据序列即是磁盘上的数据块. 运用交互式定理证明工具 Athena, 他们还验证了一系列的文件操作, 不过该工作没有支持局部推导. 针对 POSIX 文件系统, Freitas 等人通过 Z/Eves 定理证明器进行建模, 分析和验证了一个文件存储库的高安全性[38]. 同样, 针对 POSIX 文件系统, Gardner 等人提出了一种支持局部推导的证明系统. 基于分离逻辑的思想, 他们将 POSIX 的文件系统抽象为文件堆和文件标识堆, 分别描述文件的内容和状态. 他们还证明了该模型中的公理与 POSIX 中的实际标准能够相对应[39]. 此外,

Chen 等人开发的文件系统 FSCQ 可以保证在系统出现崩溃时, 系统能够正确恢复从而不丢失数据[40]. 为此, 他们通过扩展 Hoare 逻辑, 提出了“Crash Hoare Logic (CHL)”作为理论框架. CHL 中的规范不仅需要描述程序执行的前后置条件, 还要描述如果发生可能的崩溃时, 系统状态性质应该符合的不变量, 亦即无论崩溃发生与否都成立的断言. 因此假如系统突然宕机, 用户可以按照相应的不变量选择对应的系统状态来恢复执行. CHL 的规范也广泛使用了分离逻辑的方式, 这大大提高了实际验证时的自动化. 这项工作的重点是对崩溃的处理机制, 不过本文考虑的是首先要如何建模与验证 CBS.

相对于传统的存储系统, 云存储的架构更为复杂. 这为验证云存储的可靠性带来了挑战, 很难直接运用已有的技术. 目前在云存储可靠性的形式化验证领域, 已经出现了很多基于定理证明的优秀结果. Blanchard 等人运用 Frama-C 软件验证工具和 Coq, 建模分析了云管理程序 Anaxagoros 以及资源的隔离和保护机制[41]. Anaxagoros 是一种支持虚拟化操作系统内存的技术, 它也被用于云环境的管理中. Bobba 等人提出了运用重写系统以及配套工具 Maude, 以形式化设计云存储系统, 确保云存储可以满足给定的数据一致性和性能要求[42]. Pereverzeva 等人基于 Event-B 的方法, 形式构建了弹性数据的存储模型. 这项工作充分讨论了云存储系统的并发机制, 他们通过分析备份数据的预写日志, 形式化验证了各备份数据间的数据一致性问题, 并且考虑了同步、半同步和异步的 3 种不同架构中, 数据的完整性和一致性[43].

一般来讲, 运用纸笔推导验证程序可能会出现疏漏, 同时又需要大量的人力成本. 并且, 由于证明过程的可读性较差, 导致研究人员很难进行核查. 为了增强理论模型的实用性, 以及适应不同的程序验证场景, 诸多基于分离逻辑的优秀验证工具应运而生.

Appel 团队运用分离逻辑, 在 Coq 中形式化建模了 Cminor[44,45], 该工作延伸出了支持验证 C 语言程序的工具 VST[46]. 这个工具由曹钦翔等人开发, 它能够验证并发 C 语言程序的安全性以及功能的正确性. VST 支持推理实际的 C 语言程序, 以及验证列表、堆栈、散列表等复杂的数据结构. 曹钦翔等人还为 VST 出版了教程, 刊载“软件基础(software foundations)”系列的第 5 卷[47].

邵中团队实现的 CAP 框架支持验证动态内存操作的底层代码[48]. 对 CAP 框架进行拓展, Ni 等人开发的 XCAP 可以验证 x86 系统中带有上下文的管理程序[49]; 冯新宇等人提出的 SCAP 可以推理基于堆栈的控制程序[50], 同样, 由他们提出的 OCAP 支持验证不同系统的互操作性[51]. 基于 SCAP 和 OCAP, 冯新宇等人还验证了运用硬件中断的线程抢占式调度[52].

詹博华在 Isabelle 中开发的自动推理工具 Auto2, 基于最佳优先(best-first)的搜索方式, 并允许用户添加启发式的搜索策略[53]. Auto2 支持在证明过程中, 通过搜索自动且持续地解决各种子问题, 从而实现用最为简洁的证明脚本完成对程序的形式化验证.

此外, 还有很多基于分离逻辑的优秀工具: Jung 等人开发的 Iris 框架, 支持高阶的并发分离逻辑[54]. Bengtson 等人开发的 Charge!工具, 支持验证面向对象的程序设计语言[55]. Nanevski 等人开发的 Ynot, 支持验证依赖类型系统的函数式语言[56]. Chlipala 等人开发的 Bedrock, 支持验证多线程协同和 XML 处理器的执行[57]. Calcagno 等人开发的 Infer, 被应用于 Facebook 的项目开发中, 能够实现对 Java 以及 C 语言的程序进行内存检测[58].

7 总结与展望

为提高块云存储系统的可靠性, 本文从验证它的管理程序正确性入手, 在交互式定理证明器 Coq 中, 实现了一个验证工具. 基于分离逻辑的思想, 工具中证明系统的实现主要围绕如下几点展开: 抽象块云存储系统为两层结构; 构建表示实际程序的建模语言; 定义描述系统状态性质的 CBS 堆谓词; 定义描述程序行为的 CBS 分离逻辑三元组; 以及制定和证明一系列的推理规则. 最后, 结合移动数据块、读取文件内容等实例, 本文展示了工具对实际管理程序的表达和推理方式.

然而, 目前工作也存在一定的不足: 对于云存储系统的建模来讲, 本文考虑了最基本的架构, 即不备份数据块的情况, 而实际上, 云存储系统会通过备份数据块来增强容错性, 这会引发多副本间的数据一致性问

题, 因此我们有必要考虑对云存储系统并发操作的验证. 目前验证传统并发文件系统的进展十分活跃, 比如浙江大学陈海波团队提出的 AtomFS, 就是第一个形式化验证的并发文件系统[59]. 该工作基于并发程序精化验证的相关理论, 通过分析文件并发操作在多核硬件中的线性一致性, 证明了 AtomFS 的操作接口在任意并发环境下的功能正确性和原子性. 这项工作对于验证云存储的并发机制具有一定的借鉴意义, 如果能在云存储系统建模中引入多副本, 从并发操作的角度, 讨论逻辑存储的安全性和多副本间的数据一致性, 就能更好地提高云存储系统的可靠性.

此外, 对于工具本身来讲, 我们需要引入While循环以及相应的循环不变量, 从而提高对循环程序的验证能力, 以验证 CBS 中更广泛的数据操作场景; 其次, 还需要更多的证明和搜索策略, 并引入最弱前置条件来实现递归推导, 以此提高自动化证明能力. 诸如上述的不足, 我们将在后期工作中作进一步的探索和研究.

References:

[1] Market and Market. Cloud computing market. 2020. https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html

[2] Xinhua. The 14th five-year plan for national economic and social development and the long-range objectives through the year 2035. 2021 (in Chinese). http://www.gov.cn/xinwen/2021-03/13/content_5592681.htm

[3] Mesbahi MR, Rahmani AM, Hosseinzadeh M. Reliability and high availability in cloud computing environments: A reference roadmap. Human-centric Computing and Information Sciences, 2018, 8(1): 20−51.

[4] Tung L. Amazon: Here's what caused the major AWS outage last week. 2020. https://www.zdnet.com/article/amazon-heres-what-caused-major-aws-outage-last-week-apologies/

[5] Gawanmeh A, Alomari A. Challenges in formal methods for testing and verification of cloud computing systems. Scalable Computing: Practice and Experience, 2015, 16(3): 321−332.

[6] IBM Cloud Education. What is block storage? 2019. https://www.ibm.com/cloud/learn/block-storage

[7] Mike M, Gregory RG, Erik R. Object-based storage. IEEE Communications Magazine, 2003, 41(8): 84−90.

[8] Newcombe C, Rath T, Zhang F, Munteanu B, Brooker M, Deardeuff M. How Amazon Web services uses formal methods. Communications of the ACM, 2015, 58(4): 66−73.

[9] Wing JM. A specifier's introduction to formal methods. Computer, 1990, 23(9): 8−22.

[10] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 33−61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[11] Gallier JH. Logic for Computer Science: Foundations of Automatic Theorem Proving. Dover Publications, 2015.

[12] Loveland DW. Automated Theorem Proving: A Logical Basis. Elsevier, 2016.

[13] Fitting M. First-order Logic and Automated Theorem Proving. Springer Science & Business Media, 2012.

[14] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th IEEE Symp. on Logic in Computer Science. 2002. 55−74.

[15] O'Hearn PW. Separation logic. Communications of the ACM, 2019, 62(2): 86−95.

[16] Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576−580.

[17] Qin SC, Xu ZW, Ming Z. Survey of research on program verification via separation logic. Ruan Jian Xue Bao/Journal of Software, 2017, 28(8): 2010−2025 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5272.htm [doi: 10.13328/j.cnki.jos.005272]

[18] Huang DM, Zeng QK. Program verification techniques based on separation logic. Ruan Jian Xue Bao/Journal of Software, 2009, 20(8): 2051−2061 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/2051.htm

[19] Pym D, Spring JM, O'Hearn PW. Why separation logic works. Philosophy & Technology, 2019, 32(3): 483−516.

[20] The Coq proof assistant. https://coq.inria.fr/

[21] Jiang N, Li QA, Wang LM, Zhang XT, He YX. Overview on mechanized theorem proving. Ruan Jian Xue Bao/Journal of Software, 2020, 31(1): 82−112 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5870.htm [doi: 10.13328/j.cnki.jos.005870]

[22] Charguéraud A. Program verification through characteristic formulae. In: Proc. of the 16th ACM SIGPLAN Int'l Conf. on Functional Programming. 2010. 321−332.

[23] Charguéraud A. Characteristic formulae for the verification of imperative programs. In: Proc. of the 17th ACM SIGPLAN Int'l Conf. on Functional Programming. 2011. 418−430.

[24] Charguéraud A. Higher-order representation predicates in separation logic. In: Proc. of the 5th ACM SIGPLAN Conf. on Certified Programs and Proofs. 2016. 3−14.

[25] Charguéraud A, Pottier F. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning, 2019, 62(3): 331−365.

[26] Charguéraud A. Characteristic formulae for mechanized program verification [Ph.D. Thesis]. Université Paris, 2010.

[27] Charguéraud A. Separation logic for sequential programs (functional pearl). Proc. of the ACM on Programming Languages, 2020, 116(4): 1−34.

[28] Charguéraud A. Software Foundations, Vol.6: Separation Logic Foundations. Electronic Textbook, 2021.

[29] Apache. HDFS architecture guide. 2020. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[30] Jing YX, Wang HP, Huang Y, Zhang L, Xu J, Cao YZ. A modeling language to describe massive data storage management in cyber-physical systems. Journal of Parallel and Distributed Computing, 2017, 103: 113−120.

[31] Jin Z, Wang HP, Zhang BW, Zhang L, Cao YZ. Reasoning about cloud storage systems based on separation logic. Chinese Journal of Computers, 2020, 43(12): 2227−2240 (in Chinese with English abstract).

[32] White T. Hadoop: The Definitive Guide. O'Reilly Media Inc., 2012.

[33] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Proc. of the 19th ACM Symp. on Operating Systems Principles. 2003. 29−43.

[34] Apache. HDFS federation. 2016. https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/Federation.html

[35] Apache. Disk balancer. 2017. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSDiskbalancer.html

[36] Baier C, Katoen JP. Principles of Model Checking. MIT Press, 2008.

[37] Arkoudas K, Zee K, Kuncak V, Rinard M. Verifying a file system implementation. In: Proc. of the Int'l Conf. on Formal Engineering Methods. 2004. 373−390.

[38] Freitas L, Woodcock J, Fu Z. POSIX file store in Z/Eves: An experiment in the verified software repository. Science of Computer Programming, 2009, 74(4): 238−257.

[39] Gardner P, Ntzik G, Wright A. Local reasoning for the POSIX file system. In: Proc. of the European Symp. on Programming Languages and Systems. 2014. 169−188.

[40] Chen HG, Ziegler D, Chajed T, Chlipala A, Kaashoek MF, Zeldovich N. Using Crash Hoare logic for certifying the FSCQ file system. In: Proc. of the 25th Symp. on Operating Systems Principles. 2015. 18−37.

[41] Blanchard A, Kosmatov N, Lemerre M, Loulergue F. A case study on formal verification of the anaxagoros hypervisor paging system with Frama-C. In: Proc. of the Int'l Workshop on Formal Methods for Industrial Critical Systems. 2015. 15−30.

[42] Bobba R, Grov J, Gupta I, Liu S. Survivability: Design, formal modeling, and validation of cloud storage systems using Maude. Assured Cloud Computing, 2018, 10−48.

[43] Pereverzeva I, Laibinis L, Troubitsyna E, Holmberg M, Pöri M. Formal modelling of resilient data storage in cloud. In: Proc. of the Int'l Conf. on Formal Engineering Methods. 2013. 363−379.

[44] Appel AW, Blazy S. Separation logic for small-step Cminor. In: Proc. of the Int'l Conf. on Theorem Proving in Higher Order Logics. 2007. 5−21.

[45] Appel AW. Program Logics for Certified Compilers. Cambridge: Cambridge University Press, 2014.

[46] Cao QX, Beringer L, Gruetter S, Dodds J, Appel AW. VST-Floyd: A separation logic tool to verify correctness of C programs. Journal of Automated Reasoning, 2018, 61(1): 367−422.

[47] Appel AW, Cao QX. Software Foundations, Vol.5: Verifiable C. Electronic Textbook, 2020.

[48] Yu D, Hamid NA, Shao Z. Building certified libraries for PCC: Dynamic storage allocation. Science of Computer Programming, 2004, 50(1-3): 101−127.

[49] Ni ZZ, Shao Z. Certified assembly programming with embedded code pointers. In: Proc. of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2006. 320−333.

[50] Feng XY, Shao Z, Vaynberg A, Xiang S, Ni ZZ. Modular verification of assembly code with stack-based control abstractions. ACM SIGPLAN Notices, 2006, 41(6): 401−414.

[51] Feng XY, Ni ZZ, Shao Z, Guo Y. An open framework for foundational proof-carrying code. In: Proc. of the 2007 ACM SIGPLAN Int'l Workshop on Types in Languages Design and Implementation. 2007. 67−78.

[52] Feng XY, Shao Z, Dong Y, Guo Y. Certifying low-level programs with hardware interrupts and preemptive threads. ACM SIGPLAN Notices, 2008, 43(6): 170−182.

[53] Zhan BH. AUTO2, a saturation-based heuristic prover for higher-order logic. In: Proc. of the Int'l Conf. on Interactive Theorem Proving. 2016. 441−456.

[54] Jung R, Krebbers R, Jourdan JH, Bizjak A, Birkedal L, Dreyer D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, 2018, 28.

[55] Bengtson J, Jensen J B, Sieczkowski F, Birkedal L. Verifying object-oriented programs with higher-order separation logic in Coq. In: Proc. of the Int'l Conf. on Interactive Theorem Proving. 2011. 22−38.

[56] Nanevski A, Morrisett G, Shinnar A, Govereau P, Birkedal L. Ynot: Dependent types for imperative programs. In: Proc. of the 13th ACM SIGPLAN Int'l Conf. on Functional Programming. 2008. 229−240.

[57] Chlipala A. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In: Proc. of the 18th ACM SIGPLAN Int'l Conf. on Functional programming. 2013. 391−402.

[58] Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn PW, Papakonstantinou I, Purbrick J, Rodriguez D. Moving fast with software verification. In: Proc. of the NASA Formal Methods Symp. 2015. 3−11.

[59] Zou M, Ding HR, Du D, Fu M, Gu RH, Chen HB. Using concurrent relational logic with helpers for verifying the AtomFS file system. In: Proc. of the 27th Symp. on Operating Systems Principles. 2015. 259−274.

附中文参考文献:

[2] 新华社. 中华人民共和国国民经济和社会发展第十四个五年规划和 2035 年远景目标纲要. 2021. http://www.gov.cn/xinwen/2021-03/13/content_5592681.htm

[10] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33−61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[17] 秦胜潮, 许智武, 明仲. 基于分离逻辑的程序验证研究综述. 软件学报, 2017, 28(8): 2010−2025. http://www.jos.org.cn/1000-9825/5272.htm [doi: 10.13328/j.cnki.jos.005272]

[18] 黄达明, 曾庆凯. 基于分离逻辑的程序验证技术. 软件学报, 2009, 20(8): 2051−2061. http://www.jos.org.cn/1000-9825/2051.htm

[21] 江南, 李清安, 汪吕蒙, 张晓瞳, 何炎祥. 机械化定理证明研究综述. 软件学报, 2020, 31(1): 82−112. http://www.jos.org.cn/1000-9825/5870.htm [doi: 10.13328/j.cnki.jos.005870]

[31] 金钊, 王捍贫, 张博闻, 张磊, 曹永知. 基于分离逻辑的云存储系统验证. 计算机学报, 2020, 43(12): 2227−2240.

张博闻(1995−), 男, 博士生, CCF 学生会员, 主要研究领域为计算机程序的形式化验证, 交互式定理验证工具.

王捍贫(1964−), 男, 博士, 教授, 博士生导师, 主要研究领域为程序逻辑, 程序语义, 计算机系统的描述与验证.

金钊(1990−), 男, 博士生, 主要研究领域为云存储程序的推理验证, 可计算理论.

曹永知(1974−), 男, 博士, 教授, 博士生导师, CCF 专业会员, IEEE 高级会员, 主要研究领域为形式化方法及其应用, 隐私性与安全性, 不确定性推理.