Chinese Journal of Electronics
vol. 33, no. 1, pp. 112–127, January 2024
https://doi.org/10.23919/cje.2022.00.116

RESEARCH ARTICLE

Formal Verification of Data Modifications in Cloud Block Storage Based on Separation Logic

Bowen ZHANG[1], Zhao JIN[3,1], Hanpin WANG[2,1], and Yongzhi CAO[1]

1. *Key Laboratory of High Confidence Software Technologies* (*MOE*), *School of Computer Science*, *Peking University*, *Beijing 100871*, *China*
2. *School of Computer Science and Cyber Engineering*, *Guangzhou University*, *Guangzhou 510006*, *China*
3. *School of Computer and Artificial Intelligence*, *Zhengzhou University*, *Zhengzhou 450001*, *China*

Corresponding author: Hanpin WANG, Email: whpxhy@pku.edu.cn
Manuscript Received May 3, 2022; Accepted October 8, 2022

Abstract — Cloud storage is now widely used, but its reliability has always been a major concern. Cloud block storage (CBS) is a famous type of cloud storage. It has the closest architecture to the underlying storage and can provide interfaces for other types. Data modifications in CBS have potential risks such as null reference or data loss. Formal verification of these operations can improve the reliability of CBS to some extent. Although separation logic is a mainstream approach to verifying program correctness, the complex architecture of CBS creates some challenges for verifications. This paper develops a proof system based on separation logic for verifying the CBS data modifications. The proof system can represent the CBS architecture, describe the properties of the CBS system state, and specify the behavior of CBS data modifications. Using the interactive verification approach from Coq, the proof system is implemented as a verification tool. With this tool, the paper builds machine-checked proofs for the functional correctness of CBS data modifications. This work can thus analyze the reliability of cloud storage from a formal perspective.

Keywords — Formal verification, Separation logic, Cloud block storage, Verification tool, Coq.
Citation — Bowen ZHANG, Zhao JIN, Hanpin WANG, *et al.*, "Formal Verification of Data Modifications in Cloud Block Storage Based on Separation Logic," *Chinese Journal of Electronics*, vol. 33, no. 1, pp. 112–127, 2024. doi: 10.23919/cje.2022.00.116.

I. Introduction

In recent years, with the rapid development of Internet technology, cloud services have made breakthrough progress. However, with more frequent use, the reliability of cloud storage has been questioned in recent years. In the United States, nearly USD 285 million is lost per year directly caused by cloud storage failures [1]. For example, in 2020, dozens of servers in Amazon's cloud services have crashed for five hours simultaneously, which was caused by a "small increase in storage capacity" [2]. Although many providers deploy advanced failover methods, cloud storage still suffers from many service failures and reliability problems. Therefore, it is necessary to improve the reliability of cloud storage, to avoid the recurrence of such accidents and reduce the risk of data loss. The reliability involves two main dimensions: the stability of hardware infrastructure and the correctness of data management programs [3]. The management programs are directly related to users. Improving their correctness can thus improve the reliability of cloud storage. Currently, the mainstream cloud storage in the market can be classified into three categories: file storage, object storage, and block storage [4]. This classification is about the different system architectures and data storage levels. Cloud block storage (CBS) has the most direct interaction with the storage medium. It can even directly provide interfaces for the other two types [5].

Hadoop distributed file system (HDFS) [6] is one of the most representative CBS products. HDFS has a typical CBS storage approach, which allows for deploying large data sets by dividing them into discrete blocks. HDFS has a master-slave storage architecture. The master node maintains a file system, and the slave nodes store the block's actual content. The other CBS products have a similar architecture and storage method to

Associate Editor: Prof. Ju REN, Tsinghua University.

HDFS. Generally speaking, the management of data in CBS involves a logical storage relation like "file-block-content". To analyze and verify the correctness of programs in CBS, we need to pay attention to this storage relation.

Early versions of CBS products, including HDFS, follow the simple coherency model [7], which means that once a file is created, users cannot modify the file's content. That model is designed to simplify data coherency issues and enable high throughput data access. However, with the growing popularity of CBS, various usage scenarios require CBS to support the modification of existing files. Therefore, the updated CBS products started to support data modifications. HDFS typically introduces two management programs: append and truncate, to add and remove content at the end of a file. Compared to basic CBS data operations, these modifications are more complex to execute and more likely to cause logical storage errors, such as null reference or block loss. Therefore, it is necessary to analyze their correctness.

The correctness of a program generally refers to the fact that each input will produce the desired result. There are two main approaches to verifying it: testing and formal verification. The testing approach is difficult to achieve overall coverage for large systems [8]. Formal verification models complex systems as mathematical entities. It can mathematically analyze whether a program is correct [9]. Theorem proving is a formal verification technique based on mathematical logic [10]. It models a computer system as an axiomatic system with reasoning capabilities. In recent years, theorem proving has produced many results for the verification of traditional storage. But, cloud storage has a more complex architecture than local storage. In addition, the size of cloud storage may change as the requirements grow. Such factors make formal verification of cloud storage challenging, and it is also hard to directly apply the verification techniques for traditional storage. Therefore, researchers need to extend and innovate on the existing approaches. The current validation works on cloud storage mainly focus on data consistency, high availability, or integrity. However, there are fewer discussions of data operations from a logical storage perspective. Logical storage involves information to locate the target data, such as file directories and block locations. We advocate using the theorem proving technique to analyze and verify the correctness of CBS data operations, especially modifications, as a way to reduce the errors of logical storage.

Separation logic [11], [12] is a breakthrough in the area of program verification. It was originally proposed by O'Hearn and Reynolds, as a formal proof system based on Hoare logic [13]. Separation logic is an intuitive way to verify programs with explicit memory management [14]. It describes the behavior of a program t by a triple as "$\{H\}t\{Q\}$", such a triple is called a specification of t. The precondition H is a predicate that describes the input state, and the postcondition Q describes the output state. Separation logic has two features: *small-footprint specification* and *local reasoning*. According to these two features, one may verify a program by focusing on only the partial state relevant to execution without discussing the irrelevant state [15]. Thus, separation logic is suitable for verifying complex storage systems, especially for CBS.

To improve the practicality and efficiency of theoretical models, researchers usually implement a verification tool to facilitate reasoning. Coq [16] is one of the mainstream implementation environments. The proof system developed in Coq can be used as a verification tool, and the proof process of verifying a program in such a tool retains mathematical rigor. Using that tool, one can build machine-checkable proof and avoid possible negligences in pen-and-paper inference [17]. Hence, many practical tools are used for analyzing computer systems, from data structures to operating system kernels.

For analyzing and verifying CBS management programs, Zhang *et al.* implemented a verification tool [18], by directly developing a proof system in Coq. The proof system has the features of separation logic. The verification tool with Coq can encode CBS operations intuitively and verify their specifications semi-automatically. Zhang's work introduces the concepts of *immutable variables* and *mutable heap* from the functional programming language [19]. The immutable variable means that once a variable has been assigned, then its value cannot be changed. The mutable heap means that only the values allocated in the memory cell can be changed. These concepts avoid the complexity caused by mutable variables and make the code easier to reason. They thus reasoned and verified the correctness of basic data operations in CBS products, such as creation, deletion, and fetch. However, their work directly implements the proof system in Coq without giving a mathematical definition first, and it cannot verify the correctness of CBS data modifications.

In the previous work [18], they implemented a proof system directly in Coq and used Coq's built-in libraries to implement some key components. Although that approach did simplify the implementation process, the construction of their proof system was ambiguous and obscure, due to the absence of a mathematical definition. In addition, implementation in Coq directly makes their work a highly Coq-dependent tool, and all reasoning can be performed only in Coq. However, the formal verification's rigor is from that it mathematically proves the correctness of a software design. We believe that constructing a self-contained proof system with formal language alone is a necessary step. Thus, our proof system does not involve any implementation environment, and its construction is more explicit and specific. We also check whether its components are reliable. Although it is subsequently implemented as a tool to simplify reasoning, our work can improve the rigor of subsequent verification to some extent.

On the other hand, the previous work verified only the basic data operations of early CBS products, such as create, delete, and fetch. For these operations, their principle is relatively simple, their representations are more intuitive, and their verifications are smoother. The data modifications introduced later have more complex principles and are more likely to cause errors. The modifications involve some new data operations and mechanisms, such as the block append, or the truncating range of a block should not be larger than its size. Such points are the key to consider during the verification but are not covered by the previous work. We focus on these new operations, developing appropriate representations and reasoning rules, and thus build the machine-checked proofs for their functional correctness.

The main contribution of this paper is that we propose a proof system based on separation logic, which can represent, specify, and verify the CBS data modifications. According to the formal definition of this new proof system, we update our previous verification tool, which enhances the rigor of verifications in Coq. Finally, we give machine-checked proofs for the functional correctness of CBS data modifications. Our work can further improve the reliability of cloud storage from a formal perspective. The contributions are related to the following three aspects:

1) We formalize CBS data modifications and CBS state properties. In detail, the modifications involve new data operations. We introduce some primitive operations related to the key executions of modifications, including truncating and appending a block or a file. They can be organized into a compound statement to represent the actual operations. Besides, we abstract the CBS architecture as a two-tier structure. To describe the properties of a given CBS state, we define the heap predicate with mathematical language alone, instead of relying on Coq's standard libraries as in the previous work.

2) We re-formulate CBS separation logic triples to specify the behavior of a program. The semantics of such a triple reflects the details of the state, which makes the proof of reasoning rules smoother. It still follows the small-footprint specifications in traditional separation logic. Besides, we formulate reasoning rules for the new primitive operations. They can be used to verify whether a specification holds. Thus far, we give a more explicit and specific formal definition of the proof system.

3) We update the verification tool to simplify reasoning, by implementing and proving all the new definitions and rules. Using this tool, we build the machine-checked proofs of CBS modifications. Corresponding to append and truncate, we code two sample programs to represent them. These programs cover the details of actual executions, such as allowing arbitrary-sized files as input, or the truncate range of a block cannot be larger than its size. Finally, we verify the functional correctness of these example programs.

The rest of the paper is organized as follows. In Section II, the preliminaries are provided. The construction of our proof system is presented in the following three sections. The modeling language in Section III can represent the CBS architecture and data modifications, and the assertion language in Section IV can describe the properties of a given CBS state. In addition, Section V formulates the CBS separation logic triples and reasoning rules to specify and verify the programs. Then, in Section VI, a validation example of truncating a file is given. The related work is discussed in Section VII. Finally, conclusions are drawn in Section VIII, where our future research on this topic is also discussed.

The implementation of our proof system has amount to 4723 non-blank lines of Coq code. It includes 86 definitions, 318 lemmas, and the verifications of 9 usage scenarios. All definitions, lemmas, and rules in this paper are implemented and proved in Coq. We encourage the reader to check out the corresponding source files online at: https://github.com/BinksZhang/CBS-Verification.

II. Preliminaries

In this section, we review some required basic concepts and preliminaries. Represented by HDFS, this section first describes the storage architecture of CBS products, then presents the CBS data modifications in practice, and last shows how the previous work verifies the CBS basic data operations.

1. Hadoop distributed file system

Apache Hadoop is a platform providing a solution for massive amounts of data. HDFS is the core of the Hadoop storage part and provides high-throughput access for storing massive data sets. It is one of the most representative CBS products, which has characteristics like the master-slave architecture, breaking up data into blocks, and storing or operating those blocks as the basic data unit.

A typical CBS cluster comprises a single master node and multiple slave nodes [20]. In HDFS, the master node is called NameNode, and the slave nodes are called DataNodes. NameNode is the centerpiece of HDFS. It maintains a block management command set called *Block Management* and a file system called *NameSpace*. The Block Management contains block operation instructions, and the NameSpace stores the file's information such as file location, directory trees, and block metadata. In particular, the block metadata contains the key information of each block, including block location, file-to-block mappings, or block size. Each operation on files and blocks in the cluster will update the NameSpace correspondingly, especially the block metadata. On the other hand, DataNodes are physical storage to store the actual data of each block. They are responsible for block operations such as creation and deletion. After executions, they will send back the new information about blocks to NameNode for updating the NameSpace.

The data block is the unit and the physical repre-

sentation of data in HDFS. Each block is identified as a unique block location. The block size is 128 MB by default and allows users to configure it. Data blocks serve many advantages to HDFS. In particular, as being divided into blocks, a file can be larger than any single disk in an HDFS cluster.

The architecture of a classic HDFS cluster, which usually consists of one single NameNode and multiple DataNodes, is shown in Figure 1.

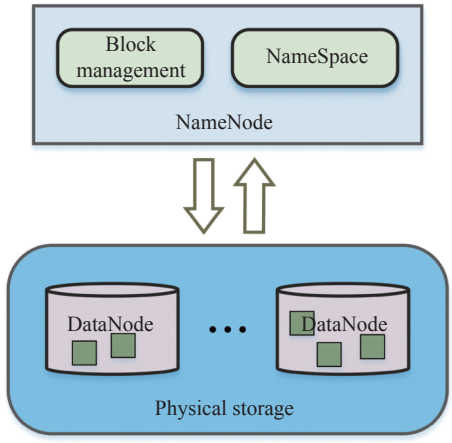

Figure 1 The storage architecture of CBS represented by HDFS.

2. Append and truncate in CBS

Early versions of CBS products follow the simple coherency model. Their applications only have write-once-read-many access for files, which means that once a file is created cannot be changed. It simplifies data coherency issues and enables high throughput data access. However, the growing popularity of CBS has made many requirements for data modifications. For example, a database may need to keep a log as an ever-expanding file. In the versions after 2.7.0, HDFS offers two additional management programs: append and truncate. Both these two modification operations are only allowed at the end of a file, not anywhere in it.

The append operation can add content at the end of a file [21]. In HDFS, a client needs to send an append request to NameNode first. Then NameNode checks the file's last block. If the last block is full, HDFS allocates a new last block. Otherwise, NameNode changes this block to be an under-construction block, writing the corresponding data into it until it is full.

The truncate operation can remove data from the end of a file [22]. After passing a file location and a truncating range into NameNode, the truncate operation directly removes all the whole blocks within the range. If the last block cannot be entirely removed, then a piece of its contents will be removed. For example, as shown in Figure 2, truncating a file f from the current length to the new length removes block3 and block4 entirely, and then it removes a piece of the contents from the tail of block2.

Other CBS products also offer these two management programs for modifications, which follow similar mechanisms, like EBS [23]. Adding and removing the contents in CBS may lead to storage errors, such as block loss or content changes accidentally. It is necessary to ensure that no such errors occur during the CBS data modifications.

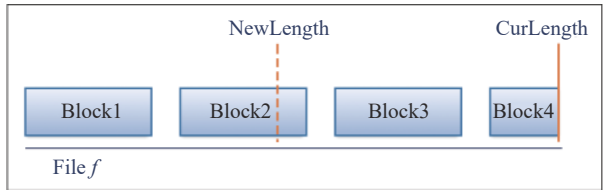

Figure 2 Truncate a file.

3. Verify CBS basic data operations

In the previous work, Zhang *et al.* implemented a verification tool for reasoning about the correctness of CBS basic data operations [18]. Based on separation logic, they developed a proof system directly in Coq. It contains a modeling language to represent the CBS architecture and operations, an assertion language to describe the properties of the CBS state, and a specification language to specify the behavior of programs.

Their proof system has the features of separation logic: *small-footprint specification* and *local reasoning*. In traditional separation logic, the behavior of a program t is specified through a triple, written as $\{H\}t\{Q\}$. Such a separation logic triple captures all the interactions that a program may have with the storage state. Any piece of state that is not described explicitly in the precondition is guaranteed to remain untouched. Zhang's proof system also encourages small-footprint specifications, i.e., specifications that mention nothing but what is strictly needed about the CBS storage state. Subsequently, local reasoning can generalize the local properties to the global state. Thus one can focus the verification only on the partial CBS state relevant to execution without discussing the irrelevant state, which is necessary to reason such a complex system as CBS.

In addition, the implementation of their proof system in Coq allows itself to be used as a verification tool. They built a series of notations to encode actual CBS operations intuitively and provided some automated proof scripts to simplify the verifying process. The mathematical rigor of that tool comes from its implementation environment Coq, which is one of the mainstream environments in implementing proof systems related to computer programs. Coq is an interactive theorem prover based on "type theory" and "inductive constructions", which constrain and strict the proof process. It uses a bottom-up reasoning approach, requiring the user to construct a derivation tree for a given proposition, that is, each premise of the proposition needs to be proven. Zhang's work makes the verification of the basic CBS operations

work smoothly. They thus verified such operations as creation, replication, fetch, and deletion.

However, their previous work implemented a proof system directly in Coq, and some components were implemented using Coq built-in libraries, such as the assertion language. Thus, the construction of their proof system was ambiguous and obscure, and the implementation in Coq directly makes their work a highly Coq-dependent tool. On the other hand, the data modifications introduced later by CBS, i.e., truncate and append, have a more complex principle and are more likely to cause errors. These modifications involve new mechanisms that the previous work cannot reason and verify.

We develop an axiomatic system with inference capabilities using the formal language alone. This construction approach makes our model a self-contained proof system, i.e., it can represent and reason CBS directly without involving any implementation environment. Besides, during the construction, we also check whether each component is reliable. Now, the construction of our proof system is more explicit and specific, which can improve the rigor of verification from a formal perspective. In addition, for reasoning and verifying actual CBS data modifications, we introduce some new instructions and rules into our proof system. Further, to simplify verification and give machine-checked proofs, we update the previous tool in Coq. We also code two sample programs to represent the CBS modifications: append and truncate. Both cover some details of actual executions, and we build the machine-checked proofs for them.

III. Modeling Language

We consider the functional language with λ-calculus in our modeling language. The reasons are as follows: First, the CBS data modifications operate on storage cells. The concepts of immutable variables and mutable heap in functional language allow us to focus more on the operations with locations. Second, λ-calculus abstracts extraneous details away from the programming language, which may allow a concise representation of file or block operations. For example, the λ-calculus reductions can directly represent introducing arguments to a function. Last, the implementation in Coq technically depends on the form of λ-calculus.

The definition of modeling language mainly concerns the CBS heap, syntax tree, and evaluation rules. According to CBS's actual architecture, we use a two-tier heap structure to abstract its state. We also introduce a series of primitive operations to represent the key parts of CBS data modifications. Thereafter, using terms (e.g., let-bindings, conditionals.) to combine them, the compound statements can thus represent the actual CBS operations. In addition, we use operational semantics to develop the evaluation rule for each introduced syntax element. These rules indicate the update of the system state and corresponding return value. Using them, one can represent the execution process of CBS data operations.

1. CBS heaps

The master node in CBS maintains the relations between files and blocks, while the slave nodes store each block's contents. According to this structure, we subdivide the CBS storage into two tiers: a file tier and a block tier. Correspondingly, we use a file heap to represent the system state at the file tier, and it is defined as a finite map from file locations to block-location sequences. Similarly, a block heap represents the storage of each block's content, and it is defined as a finite map from block locations to integer sequences. Thus, we may abstract the CBS architecture by a two-tier heap structure, as shown in Figure 3.

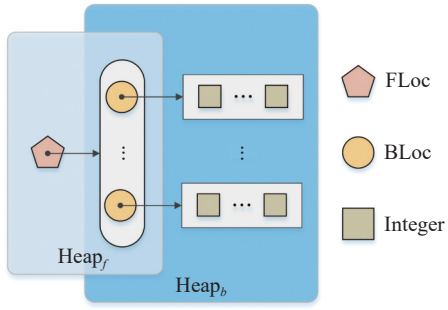

Figure 3 Using a two-tier heap structure to represent CBS state.

Thereafter, let `floc` be a file location type and `bloc` be a block location type, and they can be implemented by natural numbers. Let list(`bloc`) denote a sequence of block locations, list(`int`) denote the sequence of integers, and let `fmap` α β denote the finite maps type from α to β. The finiteness property is required to ensure that fresh locations always exist. We define the file heaps and block heaps as below.

Definition 1 (Representation of file heaps) The type of file heaps is defined as "`fmap floc` list(`bloc`)".

Definition 2 (Representation of block heaps) The type of block heaps is defined as "`fmap bloc` list(`int`)".

Let `heapf` and `heapb` denote the types of file heaps and block heaps, let h_f and h_b denote a meta-variable of these two types, respectively. Thereafter, let $h_f \perp h'_f$ assert two file heaps are disjoint, that is, there is no file location both belong to the domain of two file heaps. Let $h_f \uplus h'_f$ denote the union of two disjoint file heaps. These two operators may similarly apply to the block heap.

A CBS heap, representing a piece of CBS state, is defined by a 2-tuple consisting of a file heap and a block heap.

Definition 3 (Representation of CBS heaps) The type of CBS heaps is defined as "(`heapf` $\times$ `heapb`)".

Thereafter, let `heap` denote the type of CBS heap, and h denote a meta-variable of it. Syntactically, h may be refined as (h_f, h_b). The operators between two CBS heaps are defined by this refined version. Consider two CBS heaps h_1 and h_2, suppose they may be refined like (h_f, h_b) and (h'_f, h'_b), respectively. Let $h_1 \perp h_2$ assert that two CBS heaps are disjoint, i.e., $(h_f \perp h'_f) \wedge (h_b \perp h'_b)$.

Likewise, let $h_1 \uplus h_2$ denote the union of internal heaps in each tier, i.e., $((h_f \uplus h'_f), (h_b \uplus h'_b))$. In practice, the refined version is more effective to define the evaluations of primitive operations.

2. Types of values

Classifying the types of values can provide a protective covering that hides the underlying implementations and constrains the interaction between values [24]. The following definition of type τ contains all possible types of values in our proof system, and the implementations of these types are in Definition 5. The data type "list(A)" is the most versatile in functional programming languages [19], which can be used to store a collection of elements in type A. In particular, it simplifies our reasoning because elements in a sequence need not be expressed explicitly.

Definition 4 (Types of values)

$$\begin{array}{rcl} \tau & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{floc} \mid \texttt{bloc} \\ & & \mid \text{list}(\texttt{int}) \mid \text{list}(\texttt{bloc}) \end{array}$$

Among them, the implementation of ignorable-value type `unit` is the keyword tt; the bool type `bool` is the constants `true` and `false`; the integers `int` is implemented by the integer set. The types of file locations `floc` and block locations `bloc` are both implemented by the natural number set. The type of integer sequence `list(int)` is implemented by the power set of the integers, and the block-location sequence `list(bloc)` is the power set of the natural numbers. For simplicity's sake, we do not consider the floating-point numbers in our proof system, since we are more interested in reasoning about data management than arithmetic properties.

Definition 5 (Implementation of value types)

$$\begin{array}{llll} [\![\texttt{unit}]\!] \equiv \{tt\} & [\![\texttt{bool}]\!] \equiv \{\texttt{true}, \texttt{false}\} \\ [\![\texttt{int}]\!] \equiv \mathbb{Z} & [\![\texttt{floc}]\!] \equiv \mathbb{N} & [\![\texttt{bloc}]\!] \equiv \mathbb{N} \\ [\![\texttt{list(int)}]\!] \equiv \mathbb{Z}^* & [\![\texttt{list(bloc)}]\!] \equiv \mathbb{N}^* \end{array}$$

3. Syntax

The modeling language involves the meta-variable of value v, the file primitive operation $fprim$, the block primitive operation $bprim$, and the term t. Corresponding to the storage operations in practice, we introduce the primitive operations into $fprim$ and $bprim$. Note that the term t only represents the execution order, which is just rewritten from our previous work [18]. Besides, some other primitive operations omitted here are unnecessary for verifying modifications.

Definition 6 (Syntax)

$$\begin{array}{rcl} v & ::= & tt \mid be \mid n \mid f \mid b \mid ln \mid lb \\ fprim & ::= & \ldots \mid \texttt{attach}\ f\ lb \mid \texttt{fset}\ f\ n\ b \mid \texttt{ftrun}\ f\ n \\ bprim & ::= & \ldots \mid \texttt{bcreate}\ ln \mid \texttt{bget}\ b \\ & & \mid \texttt{append}\ b\ ln \mid \texttt{btrun}\ b\ n \\ trm & ::= & v \mid x \mid fprim \mid bprim \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\ & & \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \lambda x.t \mid \mu F.\lambda x.t \mid (t\ t) \end{array}$$

The meta-variable for value v relates to the `unit` value tt, Boolean literal be, integer n, file location f, block location b, integer sequence ln, and block-location sequence lb.

The file primitive operation $fprim$ includes: appending blocks, updating the nth block, and truncating blocks. The block primitive operation $bprim$ includes: creating a block, getting a block's contents, and appending or truncating content from a block.

The terms t include the value meta-variables v, programming variables x, file primitive operations $fprim$, block primitive operations $bprim$, conditionals, let-bindings, non-recursive functions $\lambda x.t$, recursive functions $\mu F.\lambda x.t$, and function invocations $(t\ t)$.

4. Semantics

In our proof system, the evaluation of each syntax element depends on a given full state. We use s to represent a meta-variable of type `heap` corresponding to the *full CBS storage state*. The semantics of our language is defined by the *big-step semantics judgment* with return values, which is a pattern like "$t/s \Downarrow v/s'$". This judgment describes a term t, starting from the state s, evaluates to a value v and terminates at the final state s'. The definition of semantics has two parts: the evaluation rules for terms and the evaluation rules for primitive operations.

The evaluation rules for terms are not related to the details of primitive operations but indicate how statements execute. They are rewritten from our previous work [18]. The value meta-variable evaluates itself. The let-bindings indicates the sequential execution in our language, and it carries the return value from the middle of execution. The second line is the evaluation rules for function invocations, which depend on a substitution "$t[v/x]$". The substitution "$t[v/x]$" replaces all x by v in t. The non-recursive functions may be invoked by substituting variables with arguments directly in t. While the recursive functions $\mu F.\lambda x.t$ additionally introduce the return value of previous execution to replace the function name F, thus avoiding possible confusion in recursive invocation. The last rule for conditional evaluates according to the value of Boolean literal be.

Definition 7 (Evaluation rules for terms)

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

$$\frac{(t[v/x])/s \Downarrow v'/s'}{((\lambda x.t)\ v)/s \Downarrow v'/s'} \qquad \frac{t_1/s \Downarrow v_1/s' \quad (t_2[v_1/x])/s' \Downarrow v/s''}{(\texttt{let}\ x = t_1\ \texttt{in}\ t_2)/s \Downarrow v/s''}$$

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

Thereafter, we focus on the introduced file and block primitive operations. Considering that these operations only occur at the corresponding tier, we first need to refine the representation of the CBS state. Likewise, we introduce the meta-variables s_f and s_b of type `heapf` and

heapb, to represent the full storage state in file and block tiers, respectively. Then, a given full CBS state may be represented as (s_f, s_b).

The semantics of storage primitive operations is related to updating maps and list operations. For ease of illustration, we first agree on some notations. With respect to updating maps, for each finite map m, its domain is denoted by "dom(m)", the result of mapping p in m is denoted by "$m[p]$", and updating a link from p to v in m is denoted by "$m[p := v]$". With respect to list operations, for each finite sequence l, appending l_2 to the end of l_1 is denoted by "$l_1 \cdot l_2$", updating the i-th element in l to v is denoted by "`update` i v l", and removing the last n elements in l is denoted by "`droplast` nl". The evaluation rules for the new primitive modifications are shown in Definition 8.

The evaluation of primitive operations will only update the corresponding internal state. The first three lines indicate the modifications of a file: Attaching a new block sequence to a file, with the file location f and block location sequence lb, appends lb at the end of that file's mapping result in s_f. Updating the nth block's location in a file changes the nth element in the f mapping result, thus updating s_f and returning the ignorable value tt. Truncating a file, with a location f and an integer n, removes the last n elements in the f mapping result, then updating s_f and returning tt. Although these modifications do not need to return an explicit value, they also return an ignorable value of the type `unit`.

Definition 8 (Evaluation rules for primitive operations)

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

$$\frac{f \in \mathrm{dom}(s_f)}{(\texttt{fset}\ f\ n\ b)/(s_f, s_b) \Downarrow tt/(s_f[f := (\texttt{update}\ n\ b\ s_f[f])], s_b)}$$

$$\frac{f \in \mathrm{dom}(s_f)}{(\texttt{ftrun}\ f\ n)/(s_f, s_b) \Downarrow tt/(s_f[f := (\texttt{droplast}\ n\ s_f[f])], s_b)}$$

$$\frac{b \notin \mathrm{dom}(s_b)}{(\texttt{bcreate}\ ln)/(s_f, s_b) \Downarrow b/\ (s_f, s_b[b := ln])}$$

$$\frac{b \in \mathrm{dom}(s_b)}{(\texttt{bget}\ b)/(s_f, s_b) \Downarrow (s_b[b])/(s_f, s_b)}$$

$$\frac{b \in \mathrm{dom}(s_b)}{(\texttt{append}\ b\ ln)/(s_f, s_b) \Downarrow tt/\ (s_f, s_b[b := (s_b[b] \cdot ln)])}$$

$$\frac{b \in \mathrm{dom}(s_b)}{(\texttt{btrun}\ b\ n)/(s_f, s_b) \Downarrow tt/(s_f, s_b[b := (\texttt{droplast}\ n\ s_b[b])])}$$

The following four lines indicate the operations of a block: Creating a block with an integer sequence ln returns a new block location b and creates a link from b to ln in s_b. Reading a block by the location b returns the integer sequence stored in that block, and it does not change the state. Appending new content ln to a block appends that integer sequence to the end of the mapping result, returns the value tt, and updates the map s_b. Truncating a block removes the last n elements from the tail of the mapping result, returns the value tt, and updates s_b correspondingly.

Using these evaluation rules, we can represent the execution process of CBS data operations. For example, the following function invocation in Example 1 can represent the operation of copying a block. By introducing a specific block location b_0, the invocation first replaces the function variable blk. Then, using the evaluation rule of let-binding, the function body can be subdivided as the sequential execution of two primitive operations, i.e., reading the target block's content and then creating a new block with that. This function invocation returns the new block's location b_1, and updates the block heap as $s_b[b := ln]$, which is denoted by s'_b.

Example 1 (Representation of copying a block)

$$\frac{(\texttt{bget}\ b_0)/(s_f, s_b)\ \Downarrow\ ln/(s_f, s_b) \qquad (\texttt{bcreate}\ ln)/(s_f, s_b)\ \Downarrow\ b_1/(s_f, s'_b)}{\Big(\texttt{let}\ con = (\texttt{bget}\ b_0)\ \texttt{in}\ (\texttt{bcreate}\ con)\Big)/(s_f, s_b)\ \Downarrow\ b_1/(s_f, s'_b)}$$

$$\frac{\Big(\texttt{let}\ con = (\texttt{bget}\ b_0)\ \texttt{in}\ (\texttt{bcreate}\ con)\Big)/(s_f, s_b)\ \Downarrow\ b_1/(s_f, s'_b)}{\Big((\lambda bk.\ \texttt{let}\ con = (\texttt{bget}\ bk)\ \texttt{in}\ (\texttt{bcreate}\ con))\ b_0\Big)/(s_f, s_b)\ \Downarrow\ b_1/(s_f, s'_b)}$$

IV. Assertion Language

The assertion language is used to describe the properties of a given CBS system state, which is necessary to specify a program's behavior. This section defines the CBS heap predicates to describe such properties, the entailment relation to state the logical order between predicates, and the postconditions to describe the properties of the output state and output value.

1. CBS heap predicates

Based on the higher-order predicate BI [25] in separation logic, we define the CBS heap predicate as a proposition about CBS heaps. The meaning of such a predicate is a set of CBS heaps that can satisfy the proposition. The definition of CBS heap predicates is inductive, which means we define each element that will be involved first, and then define the CBS heap predicate entirely.

In the beginning, we introduce an assignment η to define the meaning of the meta-variable for each value, except bool. Because the bool meta-variable evalutes either true or flase, which depend on comparisons in the context. The assignment η maps each meta-variable to a concrete value corresponding to its type. These mapping values involve the integer n, natural number l, integer sequence instance $\bar{n}$, and natural-number sequence instance $\bar{l}$. We use $[\![v : \tau]\!]_\eta$ denote the mapping result in an assignment η from a meta-variable v. For example, for some meta-variable b in the block-location type `bloc`, since the type `bloc` is implemented by natural numbers,

then $[\![b : \texttt{bloc}]\!]_\eta$ is a concrete natural number l.

Definition 9 (Implementation of value meta-variables)

$$[\![tt : \texttt{unit}]\!]_\eta \equiv tt \qquad [\![n : \texttt{int}]\!]_\eta \equiv n$$
$$[\![b : \texttt{bloc}]\!]_\eta \equiv l \qquad [\![f : \texttt{floc}]\!]_\eta \equiv l$$
$$[\![lb : \texttt{list(bloc)}]\!]_\eta \equiv \bar{l} \qquad [\![ln : \texttt{list(int)}]\!]_\eta \equiv \bar{n}$$

Then, we define pure propositions to describe the properties of the meta-variables, which can introduce some auxiliary information, like the properties of return value in the postcondition. The pure propositions hold only with respect to the assignment and are independent of any CBS heap. They contain an equivalence comparison "$v_1 =_\tau v_2$" between two meta-variables of the same type, and a conjunction "$P_1 \wedge P_2$" between two pure propositions.

Definition 10 (Syntax of pure propositions)

$$P \quad ::= \quad v_1 =_\tau v_2 \mid P_1 \wedge P_2$$

Whether a pure proposition holds is only related to the assignment η, and its semantics is a set over the CBS heaps. Let $\mathbb{H}$ denote a universal set that consists of all CBS heaps. For an equivalence comparison, if two meta-variables take the same value, then the evaluation of this proposition is the set $\mathbb{H}$, otherwise is an empty set $\oslash$. Likewise, for a conjunction between two propositions, if they both holds, then the evaluation is the set $\mathbb{H}$, otherwise is an empty set.

Definition 11 (Semantics of pure propositions)

$$[\![\ v_1 =_\tau v_2\ ]\!]_\eta \equiv \begin{cases} \mathbb{H}, & \text{if } [\![v_1 : \tau]\!]_\eta = [\![v_2 : \tau]\!]_\eta \\ \oslash, & \text{otherwise} \end{cases}$$
$$[\![\ P_1 \wedge P_2\ ]\!]_\eta \equiv \begin{cases} \mathbb{H}, & \text{if } [\![P_1]\!]_\eta = \mathbb{H} \text{ and } [\![P_2]\!]_\eta = \mathbb{H} \\ \oslash, & \text{otherwise} \end{cases}$$

Next, as mentioned above, the primitive operations update the system state at the internal storage level, so we need to describe the system state properties in detail. For the internal storage state, we define the file heap predicates and block heap predicates to describe their properties, respectively. The notation T means that the proposition of heaps holds and F means that it does not.

Definition 12 (File heap precidates) A file heap predicate is a predicate of type "$\texttt{heapf} \to \{T, F\}$".

Definition 13 (Block heap precidates) A block heap predicate is a predicate of type "$\texttt{heapb} \to \{T, F\}$".

We let H_f denote a file heap predicate and H_b denote a block heap predicate. Since describing the different internal heaps, these two predicates have different types and need to be defined individually. The syntax of them is shown below.

Definition 14 (Syntax of internal heap predicates)

$$H_f \quad ::= \quad [\,]_f \mid f \mapsto_f lb \mid H_f \star_f H'_f$$
$$H_b \quad ::= \quad [\,]_b \mid b \mapsto_b ln \mid H_b \star_b H'_b$$

The internal heap predicates consist of the file and block heap predicates. The file heap predicates are empty file heap $[\,]_f$, file singleton $f \mapsto_f lb$, and file separating conjunction $H_f \star_f H'_f$. Likewise, the block heap predicates are empty block heap $[\,]_b$, block singleton $b \mapsto_b ln$, and block separating conjunction $H_b \star_b H'_b$.

Internal heap predicates describe the properties of the internal state, and their semantics is a set consisting of the corresponding internal heaps. For example, a file heap predicate is a proposition about the file-tier state, and its semantics under an assignment η is the set of file heaps that can satisfy the proposition. The semantics of internal heap predicates is defined below.

Definition 15 (Semantics of internal heap predicates)

$$[\![\ [\,]_f\ ]\!]_\eta \equiv \{h_f \mid \mathrm{dom}(h_f) = \oslash\}$$
$$[\![\ f \mapsto_f lb\ ]\!]_\eta \equiv \{h_f \mid \mathrm{dom}(h_f) = [\![f]\!]_\eta \text{ and } h_f([\![f]\!]_\eta) = [\![lb]\!]_\eta\ \}$$
$$[\![\ H_f \star_f H'_f\ ]\!]_\eta \equiv \{h_f \mid \exists h_f^1 h_f^2.\ (h_f^1 \perp h_f^2)\ \wedge\ (h_f = h_f^1 \uplus h_f^2)\ \wedge\ (h_f^1 \in [\![H_f]\!]_\eta)\ \wedge\ (h_f^2 \in [\![H'_f]\!]_\eta)\ \}$$
$$[\![\ [\,]_b\ ]\!]_\eta \equiv \{h_b \mid \mathrm{dom}(h_b) = \oslash\}$$
$$[\![\ b \mapsto_b ln\ ]\!]_\eta \equiv \{h_b \mid \mathrm{dom}(h_b) = [\![b]\!]_\eta \text{ and } h_b([\![b]\!]_\eta) = [\![ln]\!]_\eta\ \}$$
$$[\![\ H_b \star_b H'_b\ ]\!]_\eta \equiv \{h_b \mid \exists h_b^1 h_b^2.\ (h_b^1 \perp h_b^2)\ \wedge\ (h_b = h_b^1 \uplus h_b^2)\ \wedge\ (h_b^1 \in [\![H_b]\!]_\eta)\ \wedge\ (h_b^2 \in [\![H'_b]\!]_\eta)\ \}$$

The empty file heap $[\,]_f$ characterizes the domain of a file heap as empty. Notice that this set it describes is different from the empty set. Because in the former set, one can find a mapping whose domain is empty, while the empty set does not have any elements. The file singleton $f \mapsto_f lb$ characterizes a singleton map. The map's domain only has a file location $[\![f]\!]_\eta$ and takes the value as a block sequence $[\![lb]\!]_\eta$. The file singleton can also show that the block sequence is not stored in a null file location. The file separating conjunction characterizes the file heap can be partitioned into two disjoint heaps h_f^1 and h_f^2, which satisfy the two file heap predicates H_f and H'_f, respectively.

Likewise, the empty block heap characterizes that the domain of a block heap is empty. The block singleton characterizes the block heap as a singleton map from a block location $[\![b]\!]_\eta$ to an integer sequence $[\![ln]\!]_\eta$, indicating that the block location is not null. The block separating conjunction characterizes that the block heap can be partitioned into two disjoint block heaps, and these two block heaps can satisfy the corresponding block heap predicates.

Finally, we define the CBS heap predicates to describe the properties of a given CBS system state.

Definition 16 (CBS heap predicates) The type of CBS heap predicates is "$\texttt{heap} \to \{T, F\}$".

Let H denote a meta-variable in CBS heap predicate type, we define the syntax of CBS heap predicates as follows.

Definition 17 (Syntax of CBS heap predicates)

$$H ::= [\,] \mid P \mid \langle H_f, H_b\rangle \mid H_1 \star H_2 \mid \exists v:\tau.H \mid \forall v:\tau.H$$

The syntax of CBS heap predicates contains empty heap $[\,]$, pure proposition P, refinement predicate $\langle H_f, H_b\rangle$, separating conjunction $H_1 \star H_2$, existential quantifier $\exists v:\tau.H$, and universal quantifier $\forall v:\tau.H$. Note that we usually omit the types as $\exists v.H$ or $\forall v.H$ in practice reasoning since they can be inferred from the context. For a CBS heap h, let $h.f$ denote its internal file heap and $h.b$ denote its block heap. The semantics of a CBS heap predicate is a set about the CBS heaps, defined as follows.

Definition 18 (Semantics of CBS heap predicates)

$$\begin{aligned}
[\![\,[\,]\,]\!]_\eta &\equiv \{h \mid \mathrm{dom}(h.f) = \oslash \ \wedge\ \mathrm{dom}(h.b) = \oslash\}\\
[\![\,\langle H_f, H_b\rangle\,]\!]_\eta &\equiv \{h \mid h.f \in [\![H_f]\!]_\eta \ \wedge\ h.b \in [\![H_b]\!]_\eta\,\}\\
[\![\,H_1 \star H_2\,]\!]_\eta &\equiv \{h \mid \exists h_1 h_2.\ (h_1 \perp h_2) \ \wedge\ (h = h_1 \uplus h_2) \ \wedge\ (h_1 \in [\![H_1]\!]_\eta) \ \wedge\ (h_2 \in [\![H_2]\!]_\eta)\}\\
[\![\,\exists v:\tau.H\,]\!]_\eta &\equiv \bigcup_{i\in[\![\tau]\!]} [\![H]\!]_{\eta[v\to i]}\\
[\![\,\forall v:\tau.H\,]\!]_\eta &\equiv \bigcap_{i\in[\![\tau]\!]} [\![H]\!]_{\eta[v\to i]}
\end{aligned}$$

The empty heap $[\,]$ characterizes a CBS heap that consists of an empty file heap and an empty block heap. The pure proposition P still follows its semantics $[\![P]\!]_\eta$ in Definition 11. The refinement predicate $\langle H_f, H_b\rangle$ characterizes that inside a CBS heap, the file heap satisfies H_f, and the block heap satisfies H_b. The separating conjunction $H_1 \star H_2$ characterizes that a CBS heap can be partitioned into two disjoint heaps h_1 and h_2, which satisfy H_1 and H_2, respectively.

The existence quantifier $\exists v:\tau.H$ describes the union of all sets that can make H hold with mapping x in η as an instance i. The universal quantifier $\forall v:\tau.H$, on the other hand, describes the intersection of all sets that can make H hold after mapping x in η as i. We write $\eta[v\to i]$ to denote that the meta-variable v maps to an instance i in the assignment η.

Remark a) For an assignment η, a CBS heap h, and a CBS heap predicate H, we let $h \vDash_\eta H$ denote $h \in [\![H]\!]_\eta$. We call $h \vDash_\eta H$ that the predicate H can be satisfied with the heap h in the assigement η.

b) Let $h \vDash H$ denote that $h \vDash_\eta H$ holds for all assignments η, and we call it that a CBS heap predicate H can be satisfied with a CBS heap h, or the heap h can satisfy the predicate H.

2. Entailment relation

Entailment is a logical order relation between heap predicates, which is necessary to construct the reasoning rules or state the properties between heap predicates. In this paper, we use "$H_1 \vdash H_2$" to denote that H_1 can entail H_2.

Definition 19 (Entailment relation) For any CBS heap predicates H_1 and H_2, if any CBS heap h satisfying H_1 also satisfies H_2, then we call H_1 can entail H_2, i.e., $H_1 \vdash H_2 \ \equiv\ \forall h.\ h \vDash H_1 \ \Rightarrow\ h \vDash H_2$.

CBS heap predicates on entailment can satisfy the reflexive, transitive, and antisymmetry properties. They correspond in turn to the following lemmas.

Lemma 1 (Logical order of CBS-heap predicates)

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

In particular, the antisymmetry property concludes on equality between two CBS heap predicates. With this property, we prove that the separating conjunction of two refinement predicates is equivalent to that of their internal heap predicates. This equivalence is crucial for reasoning in practice, since it supports the interplay between the internal state and the macro CBS state, which is illustrated as follows:

Lemma 2 (Equivalence of separating conjunction from the macro and internal perspectives)

$$\langle H_f, H_b\rangle \star \langle H'_f, H'_b\rangle \quad = \quad \langle\, (H_f \star_f H'_f),\ (H_b \star_b H'_b)\,\rangle$$

3. Generalization to CBS postconditions

According to the evaluation rules, a primitive operation will output a return value when terminated. The postcondition of a triple, called a CBS postcondition, has a type as follows. It can describe the properties of a given CBS state and additionally describes a return value.

Definition 20 (CBS postconditions) The type of CBS postconditions is "$\tau \to \mathtt{heap} \to \{T, F\}$".

According to the type theory, a CBS postcondition takes the form "$\lambda r:\tau.H$", where H is a CBS heap predicate, and r is a meta-variable bound in H to describe the return value. We can also omit it as $\lambda r.H$, since the type can be inferred during the λ-calculus reduction.

Definition 21 (λ-reduction of CBS postcondition) We let the λ-application "$(\lambda r.H)\ v$" denote introducing a concrete value meta-variable v into a CBS heap predicate H. According to λ-calculus reduction, the expression "$(\lambda r.H)\ v$" is reducible by the substitution $H[v/r]$, which replaces all meta-variable r by v in the predicate H, i.e., $(\lambda r.H)\ v \ \equiv\ H[v/r]$.

Thereafter, let Q range over CBS postconditions, i.e., Q takes the form $\lambda r.H$, then the corresponding λ-application can be written as "$Q\ v$". To define the specifications and the reasoning rules, it is convenient to ex-

tend separating conjunction and entailment for postconditions. We generalize the predicates $H \star H'$ and $H \vdash H'$ by introducing predicates $Q \dot{\star} H'$ and $Q \dot{\vdash} Q'$.

After introducing a return value, the new predicate "$Q\ v$" has a CBS heap predicate type, that is, it is a predicate that may directly apply to the separating conjunction.

Definition 22 (Separating conjunction between a postcondition and a CBS heap predicate)

$$Q \dot{\star} H' \quad \equiv \quad \lambda v.\ (Q\ v \star H)$$

The entailment relation between postconditions is an extension of the entailment between CBS heap predicates. For postcondition, it states that Q entails Q' if and only if, for any value meta-variable v, the CBS-heap predicate $Q\ v$ entails $Q'\ v$.

Definition 23 (Entailment between postconditions)

$$Q \dot{\vdash} Q' \quad \equiv \quad \forall v.\ (Q\ v \vdash Q'\ v)$$

V. Specifications

This section illustrates how to specify and verify the behavior of a program. We define a CBS separation logic triple to specify the state properties before and after program execution. Based on the concept of separation logic, the triple only needs to describe the state corresponding to the execution. We also formulate the reasoning rules about the introduced operations and rewrite the structural rules with the CBS triples form.

1. CBS separation logic triples

A CBS separation logic triple can specify the behavior of a program. Such a triple of a program t is written as "$\{H\}t\{Q\}$", which is also called a specification of t. The precondition H is a CBS heap predicate describing the input state. The postcondition Q, shaped as "$\lambda r.\ H$", describes the output state and the output value. In particular, the precondition in this triple only needs to describe the piece of CBS state which is related to the execution.

The definition of CBS separation logic triple, shown below, reads as follows: if the input state decomposes as a part h_1 that satisfies the precondition H and a disjoint part h_2 that represents the rest of state, then the evaluation of term t produces an output value v and an output CBS state $(h_1' \uplus h_2)$, as the evaluation judgement "$t/(h_1 \uplus h_2) \Downarrow v/(h_1' \uplus h_2)$"; meanwhile, the value v and heap h_1' together satisfy the postcondition Q. Notice that the output state is made of a part h_1' and, disjointly, a part h_2 which was unmodified by t.

Definition 24 (CBS separation logic triples)

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall h_1.\forall h_2.\ \left\{ \begin{array}{l} h_1 \vDash H \\ h_1 \perp h_2 \end{array} \right. \Rightarrow$$

$$\exists h_1'.\ \exists (v:\tau).\ \left\{ \begin{array}{l} h_1' \perp h_2 \\ t/(h_1 \uplus h_2) \Downarrow v/(h_1' \uplus h_2) \\ h_1' \ \vDash \ Q\ v \end{array} \right.$$

This definition fits naturally with the local reasoning since it asserts from the beginning that any resource is preserved if it is not mentioned in the precondition. In addition, as the primitive operations evaluate with the internal states, a refined triple form, written as $\{\langle H_f, H_b\rangle\}\ t\ \{\lambda r.\langle H_f', H_b'\rangle\}$, is more suitable to reason the primitive operations on the specific storage level. Notice that $\langle H_f, H_b\rangle$ is a refinement predicate in the CBS heap predicates type.

For example, the following triple specifies the behavior of the copy block operation. We use `Copy_blk` to denote the copy function in Example 1. The precondition describes that the target block is stored at location b with contents ln. The postcondition describes that at the end of a function invocation with the argument b, there is a new block stored at location b', also with contents ln.

Example 2 (Specification of copying a block)

$$\{\langle H_f, (b \mapsto_b ln)\rangle\} \quad (\texttt{Copy\_blk}\ b)$$
$$\{\lambda r.\ \exists b'.\ (r =_\tau b) \star \langle H_f, (b' \mapsto_b ln) \star_b (b \mapsto_b ln)\rangle\}$$

2. Reasoning rules

The reasoning rules in our logic fall into three categories. First, the structural rules: do not depend on the details of the programming language. Second, the reasoning rules for terms: include one such rule for each term construct of the language (like conditionals). Third, the specifications of primitive operations: include such rules for each kind of internal primitive operation.

The structural rules, shown below, include the consequence rule, frame rule, and extraction rules for pure assertions and quantifiers. They are just rewritten from traditional separation logic [11] into our proof system.

Lemma 3 (Structrual rules)

$$\frac{H \vdash H' \quad \{H'\}\ t\ \{Q'\} \quad Q' \dot{\vdash} Q}{\{H\}\ t\ \{Q\}} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \dot{\star} H'\}}$$

$$\frac{P \Rightarrow \{H\}\ t\ \{Q\}}{\{P \star H\}\ t\ \{Q\}} \qquad \frac{\forall x.\{H\}\ t\ \{Q\}}{\{\exists x.H\}\ t\ \{Q\}}$$

The consequence rule allows strengthening the precondition and weakening the postcondition. The frame rule asserts that if a term t safely executes in a given piece of state, it also can execute safely in a larger piece of state. The next two rules can extract the pure propositions and the existential quantifiers from the preconditions.

The reasoning rules for terms include one rule for each term. These rules are independent of primitive operations and are rewritten from our previous work.

Lemma 4 (Reasoning rules for terms)

$$\frac{H \vdash (Q\ v)}{\{H\}\ v\ \{Q\}} \qquad \frac{\{H\}\ t_1\ \{Q'\} \qquad \forall v.\{Q'\ v\}\ (t_2[v/x])\ \{Q\}}{\{H\}\ (\texttt{let}\ x = t_1\ \texttt{in}\ t_2)\ \{Q\}}$$

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

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

The first rule applies to the evaluation of values, and it does not change the state. The next rule applies to let-binding, which splits the proof for sequential execution into two subgoals. It brings the intermediate return value into the subsequent execution. The rules in the second line apply to function invocations. When introducing a concrete argument, one needs to prove that the triple holds after the substitution in the function body. Besides, for the invocation of a recursive function, the function name is additionally substituted with the return value of the previous execution. The last line is the rule for conditionals.

The reasoning rules for primitive operations include the specifications of the file and block primitive operations. These specifications are the rules for reasoning about primitive operations without any premises.

Lemma 5 (Specifications for primitive operations)

$$
\begin{array}{ll}
\{\langle (f \mapsto_f lb),\, H_b\rangle\} & (\texttt{attach}\ f\ lb') \quad \{\lambda r.\ (r =_\tau tt) \star \langle f \mapsto_f (lb \cdot lb'), H_b\rangle\} \\
\{\langle (f \mapsto_f lb), H_b\rangle\} & (\texttt{fset}\ f\ n\ b) \quad \{\lambda r.\ (r =_\tau tt) \star \langle f \mapsto_f (\texttt{update}\ n\ b\ lb), H_b\rangle\} \\
\{\langle (f \mapsto_f lb), H_b\rangle\} & (\texttt{ftrun}\ f\ n) \quad \{\lambda r.\ (r =_\tau tt) \star \langle f \mapsto_f (\texttt{droplast}\ n\ lb), H_b\rangle\} \\
\{\langle H_f, [\,]_b\rangle\} & (\texttt{bcreate}\ ln) \quad \{\lambda r.\ \exists b.(r =_\tau b) \star \langle H_f, b \mapsto_b ln\rangle\} \\
\{\langle H_f, (b \mapsto_b ln)\rangle\} & (\texttt{bget}\ ln) \quad \{\lambda r.\ (r =_\tau ln) \star \langle H_f, b \mapsto_b ln\rangle\} \\
\{\langle H_f, (b \mapsto_b ln)\rangle\} & (\texttt{append}\ b\ ln') \quad \{\lambda r.\ (r =_\tau tt) \star \langle H_f, b \mapsto_b (ln \cdot ln')\rangle\} \\
\{\langle H_f, (b \mapsto_b ln)\rangle\} & (\texttt{btrun}\ b\ n) \quad \{\lambda r.\ (r =_\tau tt) \star \langle H_f, b \mapsto_b (\texttt{droplast}\ n\ ln)\rangle\}
\end{array}
$$

The first three rules are the different ways of modifying a file, and all the return values of them can be ignored. The modifications of a file all require a precondition as a predicate $f \mapsto_f lb$, which describes the file cell to be manipulated. The postcondition in appending "$\texttt{attach}\ f\ lb$" asserts that the final file heap may be described by $f \mapsto_f (lb \cdot lb')$, reflecting the concatenation of two block-location sequences. The postcondition in updating the nth block "$\texttt{fset}\ f\ n\ b$" asserts that the updated file heap may be described by $f \mapsto_f (\texttt{update}\ n\ b\ lb)$, reflecting change the nth element in sequence lb as b. The postcondition in truncating "$\texttt{ftrun}\ f\ n$" asserts that the final file heap may be described by $f \mapsto_f (\texttt{droplast}\ n\ lb)$, where the new list $(\texttt{droplast}\ n\ lb)$ removes the last n block locations at the end of the sequence.

Then the next two rules are the operations of a block. Creating a block by an integer sequence "$\texttt{bcreate}\ ln$" can execute in an empty block heap, and it introduces a new block singleton cell $b \mapsto_b ln$ into the state of the block tier and returns the location of that new block. Reading a block's content "$\texttt{bget}\ b$" requires a block singleton cell as $b \mapsto_b ln$. Its postcondition asserts that the result value, named r, is the corresponding block's content ln, and this operation does not change any state.

The last two rules are the different ways of modifying a block. They all return an ignorable value and require the existence of a block cell, as $b \mapsto_b ln$. These rules assert: appending a block "$\texttt{append}\ b\ ln'$" updates the mapping result of b in the block heap to $(ln \cdot ln')$, i.e., the concatenation of two contents; moreover, truncating a block "$\texttt{btrun}\ b\ n$" modifies the block cell as $b \mapsto_b (\texttt{droplast}\ n\ ln)$, reflecting the removal of a part of block content.

VI. Verification of Modifications

In Coq, we implement the above proof system and prove the soundness of all its reasoning rules. In particular, the frame rule holds, showing that our proof system supports local reasoning for CBS programs. The implementation makes our proof system a verification tool. Using it, one can represent the CBS data operations, verify their correctness, and generate a machine-checked proof. Our new tool still supports verifying the basic CBS data operations, like proving the specification of copying a block in Example 2. Additionally, we can also use it to verify the CBS modifications, i.e., append and truncate. This section presents the verification of them to demonstrate how our tool codes and verifies actual CBS data modifications.

Three steps are required for verifying an actual CBS operation: First, code a function by the modeling language to represent the corresponding operation; Second, specify the invocation of this function by a triple; Last, reason and prove this triple using the proven reasoning rules. All steps can be directly implemented by code, which may reduce the labor costs and avoid potential negligences of pen-and-paper inference.

In addition, the files stored in the real CBS do not share data blocks, and our model also follows this principle. In our proof system, different files sharing a data block will raise logical errors. Thus, we can avoid modifying one file leads to affecting other files. In detail, when describing the overall storage of a file, one needs to characterize the file's every block. However, the repeated descriptions of the same block can cause contradictions. For example, in a given state, if two files are stored at different locations, and they both index the same block, then the state can be described like $\langle f_1 \mapsto_f b, b \mapsto_b ln\rangle \star \langle f_2 \mapsto_f b, b \mapsto_b ln\rangle$. And according to the Lemma 2, this formula is equivalent to formula $\langle (f_1 \mapsto_f b) \star_f (f_2 \mapsto_f b), (b \mapsto_b ln) \star_b (b \mapsto_b ln)\rangle$. It means that now there are two same blocks in the block storage tier. However, the definition of block separation conjunction mentions that the two heaps should be disjoint. Thus, such a formula is a contradiction, which makes it cannot a postcondition of a triple that holds. With this property, our system can

avoid multiple files sharing same blocks, thus we can focus the verification of data modifications only on the target file.

1. Verification of truncating

As mentioned before, truncating a file removes the block and content from the end of a file. This operation may lead to storage errors, which makes the verification of its correctness necessary. We represent this modification as a compound statement which consists of primitive operations. The recursive function `Truncate_File` shown in Figure 4 represents truncating an arbitrary file. Corresponding to the actual operation, this function only requires a file location and a truncating range, and it can terminate by itself. The execution of `Truncate_File` may subdivide into the following parts: First, it compares whether the range is smaller than the last block's size. If so, the function truncates the last block and ends the execution. Otherwise, the function removes the last block entirely and modifies the truncating range, then recursively invokes that function with the new range.

The declaration "`Fix F f n` $:=\cdots$" denotes that a recursive function requires two arguments: a file location f and a truncating range n. Notice that "`F`" is the function's name of the internal recursive invocation, such as "`F f n'`" in the last line. This declaration corresponds to the $(\mu F.\lambda fn.\ \cdots)$ in our proof system. By introducing the different arguments, the function invocation, shaped as "`Truncate_File f n`", may represent truncating the arbitrary file.

In our tool, we use a triple like "`triple t H Q`" to code the specification of a program, and the postcondition $\lambda r.H'$ is coded as "`(fun r => H')`". In Coq, we may use "`(fun_ => ⋯)`" to describe directly that the return value can be ignored. In addition, "`\R[Hf,Hb]`" is the refinement CBS heap predicate $\langle H_f, H_b\rangle$; the block singleton heap is "`b ∼ b ∼> ln`", and the file singleton heap is "`f ∼ f ∼> lb`"; moreover, "`Hb1 \b∗ Hb2`" is the separating conjunction between two block-heap predicates corresponding to $H_b^1 \star_b H_b^2$.

Now, consider the following storage case: a file, stored at location `f`, has three blocks, each stored at locations `p1`, `p2`, and `p3`, respectively. The file indexes these block locations as a sequence (`p1 :: p2 :: p3 :: nil`). Then, the following specification in Figure 5 may describe removing two contents from the tail of that file.

This specification characterizes the behavior of truncating: the precondition describes that the target file is stored at location f, which indexes the block-location sequence, and each block has its concrete contents. The postcondition describes that the execution returns an ignorable value. Moreover, at the block tier, this operation entirely removes the last block and partially removes the contents of the new last one; at the file tier, it updates the block-location sequence correspondingly.

To prove the specification of truncate, we first need to split the compound statement, using the reasoning rules for let-bindings or conditionals, and then prove the primitive operation of each step, using the corresponding specifications. The above proof scripts in Figure 6 is the machine-checked proof script for this specification. Since the recursive operation has some repetitive parts, we also write automated proof scripts to shorten the proof process.

```
Definition Truncate_File :=
  Fix F f n :=
(* m  is the size of the file f *)
    Let m := fsize f in
(* m1  is the index of file's tail *)
    Let m1 := m  - 1 in
(* bk  is the file's last block *)
    Let bk := nth_blk f m1 in
(* k is the size of bk *)
    Let k := bsize bk in
(* be is a bool result of 'n <= k' *)
    Let be := (n <= k) in
    If be
    Then
(* truncate n elements from the tail of block bk *)
      btrun bk n
    Else
(* n' is the new truncating range *)
      Let n'  := n - k in
(* delete the entire last block *)
      Let i1 := bdelete bk in
(* truncate the file's last block *)
      Let i2 := ftrun f 1 in
(* recursive function invocation *)
       F f n'.
```

Figure 4 Representation of truncating a file.

```
Lemma triple_Truncate: forall Hf Hb (f:floc)
                (p1 p2 p3:bloc) (n1 n2 n3 n4 n5:int),
Hf=( f ~f~> (p1::p2::p3::nil) ) ->
Hb=( (p1~b~>(n1::n2::nil))\b*(p2~b~>(n3::n4::nil))
     \b* (p3 ~b~> (n5::nil)) ) ->
triple (Truncate f 2)
  (\R[Hf,Hb])
  (fun _ => \R[ f ~f~> (p1::p2::nil),
      (p1~b~>(n1::n2::nil)) \b* (p2~b~>(n3::nil))]).
```

Figure 5 Specification of truncating a file.

2. Verification of appending

Appending is another common modification method in CBS. For example, a database may need to keep a log as an ever-expanding file, which always requires append content to that file. So we use a recursive function `Append_File` in Figure 7 to represent appending a file with arbitrary-sized content. This function needs to input a file location and appended contents, and it also can terminate by itself. The execution of `Append_File` can be subdivided as follows: First, it compares whether the appended content is smaller than the block size. If so, this function appends a block with the contents to that file. Otherwise, this function extracts parts of contents to create a block and recursively invokes that function with the rest contents.

Likewise, "`Fix F f ln`" represents a recursive function with two arguments, i.e., a file location and the appended contents. The function invocation denoted as "`Append_File f ln`" may represent appending contents

```
Proof .
(* The  first recursive execution  *)
  (* initializethe proof *)
  introv MN. subst. applys* triple_app_fix2. simpl.
  (* reason about getting file size *)
  applys triple_let triple_fsize. ext.
  (* reason about the subtraction *)
  applys triple_let triple_min. ext.
  (* reason about indexing the tail block *)
  applys triple_let triple_fget_nth_blk. ext.
  applys triple_let.
  (* reason about getting block size *)
  applys triple_conseq_frame triple_bsize.
  inner_femp_r.
  intros r. applys himpl_refl. ext'.
  (* reason about the comparison *)
  applys triple_let triple_le. ext.
  (* reason about the conditional *)
  applys triple_if. rewrite le_2_1. case_if*.
  (* reason about the subtraction *)
  applys triple_let triple_min. ext.
  applys triple_let.
  (* reason about deleting the block *)
  applys triple_conseq_frame triple_bdelete.
  (* rewrite the format *)
  rewrite hstar_sep. applys himpl_refl.
  intros r. rewrite hstar_sep, hfstar_hempty_r,
       hbstar_hempty_l.
  applys himpl_refl.
  (* reason about truncating the last block *)
  intros. simpl. applys triple_let triple_ftruncate.
  (* complete the verification of the first time *)
  intros. simpl. unfold droplast. simpl. rew_list.

(* Exit recursive execution  *)
  run2time.   (* repeat the above proof scripts *)
  (* reason about the conditional *)
  applys triple_if. rewrite le_1_2. case_if*.
  (* reason about truncating the block *)
  applys triple_conseq_frame triple_btruncate.
  (* compelete the verification *)
  rewrite hstar_sep. applys himpl_refl.
  intros r. rewrite hstar_sep, hfstar_hempty_r.
  unfold droplast. simpl. rew_list.
  rewrite hbstar_comm. applys himpl_refl.
Qed .
```

Figure 6 Proof for the specification of truncating file.

to an arbitrary file.

Consider the following storage case: a CBS file is stored at location f, and it only has one block. The block is stored as b and has content (n1 :: n2). Now, we append the contents (n3 :: n4 :: n5) to that file. The triple below in Figure 8 can specify the behavior of this modification.

The postcondition in the specification characterizes that after the append operation, the system exists two new blocks associated with the file, and their contents are (n3 :: n4) and (n5), respectively. Also, the return value of this operation can be ignored. We give proof of this specification in our tool, whose Coq script is shown in Figure 9.

VII. Related Work

In this paper, we develop a proof system for verifying the correctness of CBS modifications, and we also implement it as a verification tool to improve the practicality. We divide the discussion of related work into three categories: First, the application of theorem proving techniques on traditional storage; Second, theorem proving in cloud storage; Last, some excellent verification tools based on separation logic.

```
Definition Append_file :=
  Fix F f ln :=
(* m is the size of the appended content ln *)
    Let m := len ln in
(* compare whether m is smaller than block size *)
    Let be := (m<= 2) in
    If be
    Then
(* create a block with ln at the end of the file *)
      Append_blk f ln
    Else
(* extract the contents that can be stored in a
     block *)
      Let ln1 := hd ln in
(* ln2 is the rest of contents after extraction *)
      Let ln2 := tl ln in
(* append a block with ln1 *)
      Let r := Append_blk f ln1 in
(* recursive function invocation *)
       F f ln2.
```

Figure 7 Representation of appending a file.

```
Lemma triple__Append: forall (p1:bloc) (f:floc)
                             (n1 n2 n3 n4 n5:int),
triple (Append f (n3::n4::n5::nil))
  (\R[ f ~f~> (p1::nil), p1 ~b~> (n1::n2::nil) ])
  (fun _ => \exists p2 p3,
   \R[ f ~f~> (p1::p2::p3::nil),
       p1~b~>(n1::n2::nil) \b* p2~b~>(n3::n4::nil)
       \b*  (p3 ~b~> (n5::nil))]).
```

Figure 8 Specification of appending a file.

Formal verification uses mathematical models for analyzing computer systems to establish system correctness with mathematical rigor. Existing formal verification techniques can be divided into two categories: model checking and theorem proving. Model checking [26] can check exhaustively and automatically whether a model about a system meets a given specification. Due to the state explosion problem in model checking, the number of states is usually limited. Therefore, model checking is more suitable for characterizing the precise scenarios in storage systems, such as deadlock and data protocols. In contrast, theorem proving is a formal verification technique based on mathematical logic. It allows one to formulate an axiomatic system with derivation capabilities about the large industrial systems. Then, the desired property of a system can be stated as a mathematical theorem, typically as a logical formula, and one can verify whether the theorem is provable [10].

Theorem proving in traditional storage Previous papers have studied using the theorem proving to analyze and reason about the traditional storage and underlying file systems, which inspired us to work on verifying CBS. Chen *et al.* extended traditional Hoare logic and introduced a logic framework called "Crash Hoare Logic" [27], [28]. They developed a file system FSCQ, and it can recover the system correctly without losing data when a crash happens. Rather than handling crashes, we consider how to model and verify CBS in the first place. Arkoudas *et al.* modeled the Unix file system as a map from

```
Proof .
(* The  first recursive execution  *)
  (* initializethe proof *)
  intros. applys*triple_app_fix2. simpl.
  (* reason about  getting contents ' size *)
  applys triple_let triple_list_len.
  (* reason about  the comparison  *)
  ext. applys triple_let triple_le. ext.
  (* reason about  the conditional *)
  applys triple_if. case_if* . destruct C. auto.
  (* reason about  the extraction *)
  applys triple_let triple_list_hd. ext.
  applys triple_let triple_list_tl. ext.
  applys triple_let.
  (* reason about  appending  a new  block *)
  applys triple_conseq_frame triple_Append_blk.
  (* rewrite the format  *)
  rewrite hstar_sep. rewrite hfstar_hempty_r,
       hbstar_hempty_l.
  apply himpl_refl. intros r. simpl. apply
       himpl_refl.
  (* extract the new  block 's location as p2  *)
  intros r. rewrite hstar_hexists.
  applys triple_hexists. intros p2.
  rewrite hstar_sep, hfstar_hempty_r, hbstar_comm.

(* The  second  recursive execution  *)
  (* reason about  the recursive invocation *)
  applys*triple_app_fix2. simpl.
  (* reason about  getting contents ' size *)
  applys triple_let triple_list_len. ext.
  (* reason about  the comparison  *)
  applys triple_let triple_le. ext.
  (* reason about  the conditional *)
  applys triple_if. case_if*.
  (* reason about  the otherwise  situation *)
  2: { destruct C0. rew_list. discriminate. }
  (* reason about  appending  a new  block with  (n5) *)
  applys triple_conseq_frame triple_Append_blk.
  (* rewrite the format  and  finish the proof *)
  rewrite hstar_sep, hfstar_hempty_r,
       hbstar_hempty_l. apply himpl_refl.
  rewrite hstar_hexists. apply himpl_hexists_append.
Qed .
```

Figure 9 Proof for the specification of appending file.

file names to sequences of bytes, and they presented a correctness proof for the file system using Athena proof system [29]. In contrast, the size of a block in CBS is not fixed, and our proof system additionally supports local reasoning. Gardner *et al.* provided a program logic based on separation logic for specifying the POSIX file system [30]. That work supports local reasoning, and it uses a file heap and a file-descriptor heap to represent the content and the descriptor of a file.

Theorem proving in verifying cloud storage Cloud storage has a sophisticated architecture, and its reliability is correspondingly more difficult to be verified. The existing research works, based on theorem proving, mostly focus on the data consistency, integrity, or availability. Pereverzeva *et al.* used the write-ahead logging and Event-B to verify the data consistency in replicas, during the cloud storage operations [31]. Bobba *et al.* proposed a formal design of cloud storage to ensure data consistency and stable performance. They developed a logic framework using rewriting logic and its accompanying Maude tools [32]. Blanchard *et al.* modeled and verified the cloud hypervisor for the resource isolation and protection in the virtual memory system, using Frama-C software verification tool and Coq [33].

There are also some results for reasoning the correctness of CBS management programs, from the logical storage level. Jing *et al.* proposed a modeling language to represent the system state and data operations [34]. Based on this modeling language and applying the idea of separation logic, Jin *et al.* constructed a proof system to verify the behavior of programs [35], and they also proved the adaptation completeness of that proof system [36]. However, their works cannot be fully applied to this paper. First, they only focus on the storage security of blocks but ignore that of files. Second, the content of a data block can just be an integer. Last, one can only perform the inference on pen-and-paper. In this paper, we improve all of the above points. Our proof system additionally describes the storage state of the master node, which increases the inference capability about CBS file operations. We also expand the block content into a sequence of integers, which improves the expressive power of the modeling language. In addition, reasoning in the proof assistant Coq may avoid potential negligences of pen-and-paper derivations, and the automated scripts in the tool may also reduce labor costs. Besides, the previous work [18] by Zhang *et al.* is most related to ours, in particular, it inspires us to develop the formal definitions of our proof system. We additionally analyze and verify the data modifications on top of their work.

Verification tools Based on separation logic, the last decade has seen tremendous progress in verifying programs by tools. CFML is more related to this paper, which is a verification tool for the functional programming language. CFML can verify complex data structures such as linked lists, trees, and union-find sets [37], [38]. Likewise, it implements separation logic in Coq and uses the form of λ-calculus. Thus, we can adopt a few Coq libraries directly from CFML, mainly involving the automation scripts. Besides, the Iris [39] supports higher-order logic concurrent separation logic; the Charge! [40] supports object-oriented programming languages; the Ynot [41] supports functional languages with dependent type systems; the Bedrock [42] supports the low-level code and XML processor; and the Infer [43] is used in the software development cycle at Facebook, to verify Java and C programs.

VIII. Conclusion and future work

To improve the reliability of cloud block storage, this paper focuses on verifying the correctness of its data modifications. A proof system is developed based on separation logic, and it is subsequently implemented as a verification tool in Coq. The construction of the proof system is explicit and specific, which can enhance the rigor of the corresponding verifications in Coq. This paper illustrates that our work can represent, specify, and verify the actual CBS management programs. In particular, this paper builds machine-checked proofs for the functional correctness of CBS data modifications.

However, our current work also has some shortcomings: For the tool itself, the proof process is slightly dependent on manual debugging. The reasoning ability could be improved by writing more automated scripts. On the other hand, for modeling CBS, this paper considers the most basic case, i.e., no replica of data blocks. While in CBS, the blocks are replicated to enhance the fault tolerance, and data modifications inevitably affect data consistency. If we combine consider the security of logical storage and the consistency among multiple replicas, the reliability of CBS can be better discussed. We will further explore and study these shortcomings in our future work.

Acknowledgement

This work was supported by the National Key R&D Program of China (Grant No. 2021YFF1201102) and the National Natural Science Foundation of China (Grant Nos. 61972005, 62172016, and 61932001).

References

[1] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, "Reliability and high availability in cloud computing environments: a reference roadmap," *Human-Centric Computing and Information Sciences*, vol. 8, no. 1, article no. 20, 2018.

[2] L. Tung, "Amazon: here's what caused the major AWS outage last week," Available at: https://www.zdnet.com/article/amazon-heres-what-caused-major-aws-outage-last-week-apologies/, 2020.

[3] A. Gawanmeh and A. Alomari, "Challenges in formal methods for testing and verification of cloud computing systems," *Scalable Computing:Practice and Experience*, vol. 16, no. 3, pp. 321–332, 2015.

[4] IBM Cloud Education, "What is block storage?," Available at: https://www.ibm.com/cloud/learn/block-storage, 2019.

[5] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, 2003.

[6] K. Shvachko, H. R. Kuang, S. Radia, *et al.*, "The hadoop distributed file system," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, Incline Village, NV, USA, pp. 1–10, 2010.

[7] Apache, "HDFS architecture guide," Available at: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2020.

[8] C. Newcombe, T. Rath, F. Zhang, *et al.*, "How Amazon web services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[9] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–22, 1990.

[10] J. H. Gallier, *Logic for Computer Science*: *Foundations of Automatic Theorem Proving*, 2nd ed., Dover Publications, New York, NY, USA, pp. 1–12, 2015.

[11] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, pp. 55–74, 2002.

[12] P. O'Hearn, "Separation logic," *Communications of the ACM*, vol. 62, no. 2, pp. 86–95, 2019.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[14] S. C. Qin, Z. W. Xu, and Z. Ming, "Survey of research on program verification via separation logic," *Journal of Software*, vol. 28, no. 8, pp. 2010–2025, 2017. (in Chinese)

[15] D. Pym, J. M. Spring, and P. O'Hearn, "Why separation logic works," *Philosophy & Technology*, vol. 32, no. 3, pp. 483–516, 2019.

[16] Inria, "The Coq proof assistant," Available at: https://coq.inria.fr/, 2021.

[17] N. A. Hamid and Z. Shao, "Interfacing Hoare logic and type systems for foundational proof-carrying code," in *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, Park City, UT, USA, pp. 118–135, 2004.

[18] B. W. Zhang, Z. Jin, H. P. Wang, *et al.*, "A tool for verifying cloud block storage based on separation logic," *Journal of Software*, vol. 33, no. 6, pp. 2264–2287, 2022. (in Chinese)

[19] J. Backfield, *Becoming Functional*: *Steps for Transforming Into A Functional Programmer*. O'Reilly Media, Inc., Sebastopol, CA, USA, pp. 1–3, 2014.

[20] T. White, *Hadoop*: *The Definitive Guide*, 3rd ed., O'Reilly Media, Inc., Sebastopol, CA, USA, pp. 43–53, 2012.

[21] Apache, "Append to files in HDFS," Available at: https://issues.apache.org/jira/browse/HADOOP-1700, 2016.

[22] Apache, "HDFS truncate," Available at: https://issues.apache.org/jira/ browse/HDFS-3107, 2016.

[23] Amazon, "Amazon elastic block store," Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html, 2022.

[24] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–523, 1985.

[25] B. Biering, L. Birkedal, and N. Torp-Smith, "BI hyperdoctrines and higher-order separation logic," in *Proceedings of the 14th European Symposium on Programming*, Edinburgh, UK, pp. 233–247, 2005.

[26] C. Baier and J. P. Katoen, *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, pp. 7–16, 2008.

[27] H. G. Chen, D. Ziegler, T. Chajed, *et al.*, "Using Crash Hoare logic for certifying the FSCQ file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*, Monterey, CA, USA, pp. 18–37, 2015.

[28] H. G. Chen, T. Chajed, A. Konradi, *et al.*, "Verifying a high-performance crash-safe file system using a tree specification," in *Proceedings of the 26th Symposium on Operating Systems Principles*, Shanghai, China, pp. 270–286, 2017.

[29] K. Arkoudas, K. Zee, V. Kuncak, *et al.*, "Verifying a file system implementation," in *Proceedings of the 6th International Conference on Formal Engineering Methods*, Seattle, WA, USA, pp. 373–390, 2004.

[30] P. Gardner, G. Ntzik, and A. Wright, "Local reasoning for the POSIX file system," in *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, Grenoble, France, pp. 169–188, 2014.

[31] I. Pereverzeva, L. Laibinis, E. Troubitsyna, *et al.*, "Formal modelling of resilient data storage in cloud," in *Proceedings of the 15th International Conference on Formal Engineering Methods*, Queenstown, New Zealand, pp. 363–379, 2013.

[32] R. Bobba, J. Grov, I. Gupta, *et al.*, "Survivability: Design, formal modeling, and validation of cloud storage systems using Maude," in *Assured Cloud Computing*, R. H. Campbell, C. A. Kamhoua, and K. A. Kwiat, Eds. Wiley-IEEE Press, Hoboken, NJ, USA, pp. 10–48, 2018.

[33] A. Blanchard, N. Kosmatov, M. Lemerre, *et al.*, "A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C," in *Proceedings of the 20th International Workshop on Formal Methods for Industrial Critical Systems*, Oslo, Norway, pp. 15–30, 2015.

[34] Y. X. Jing, H. P. Wang, Y. Huang, *et al.*, "A modeling language to describe massive data storage management in cyber-

physical systems," *Journal of Parallel and Distributed Computing*, vol. 103, pp. 113–120, 2017.

[35] Z. Jin, H. P. Wang, B. W. Zhang, *et al.*, "Reasoning about cloud storage systems based on separation logic," *Chinese Journal of Computers*, vol. 43, no. 12, pp. 2227–2240, 2020. (in Chinese)

[36] Z. Jin, B. W. Zhang, L. Zhang, *et al.*, "An adaptation-complete proof system for local reasoning about cloud storage systems," *Theoretical Computer Science*, vol. 903, pp. 39–73, 2022.

[37] A. Charguéraud, "Separation logic for sequential programs (functional pearl)," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, article no. 116, 2020.

[38] A. Charguéraud, "Software foundations (6): Separation logic foundations," *Electronic Textbook*, Version 1.6 (Coq 8.17 or later), 2023-08-23.

[39] R. Jung, R. Krebbers, J. H. Jourdan, *et al.*, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, article no. e20, 2018.

[40] J. Bengtson, J. B. Jensen, F. Sieczkowski, *et al.*, "Verifying object-oriented programs with higher-order separation logic in Coq," in *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, Berg en Dal, The Netherlands, pp. 22–38, 2011.

[41] A. Nanevski, G. Morrisett, A. Shinnar, *et al.*, "Ynot: Dependent types for imperative programs," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, BC, Canada, pp. 229–240, 2008.

[42] A. Chlipala, "The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, Boston, MA, USA, pp. 391–402, 2013.

[43] C. Calcagno, D. Distefano, J. Dubreil, *et al.*, "Moving fast with software verification," in *Proceedings of the 7th International Symposium on NASA Formal Methods*, Pasadena, CA, USA, pp. 3–11, 2015.

Bowen ZHANG is a Ph.D. candidate at the School of Computer Science, Peking University, China. His current research interests include formal verification, interactive theorem proving, cloud block storage, and Coq. (Email: zhangbowen@pku.edu.cn)

Zhao JIN received the Ph.D. degree from the Peking University, China, in 2022. He is working in the School of Computer and Artificial Intelligence, Zhengzhou University, China. His current research interests include program semantics, programming logic, and formal verification. (Email: jinzhao@pku.edu.cn)

Hanpin WANG received the Ph.D. degree from the Beijing Normal University, China, in 1993. He is a Professor at the School of Computer Science, Peking University, China. His current research interests include programming logic, formal semantics, description and verification of computer systems. (Email: whpxhy@pku.edu.cn)

Yongzhi CAO received the Ph.D. degree from the Beijing Normal University, China, in 2003. He is a Professor at the School of Computer Science, Peking University, China. His current research interests include formal methods and their applications, privacy and security, and reasoning about uncertainty. (Email: caoyz@pku.edu.cn)