项目编号：T030PRP39011

上海交通大學

本科生研究计划（PRP）研究论文

（第 39 期）

论文题目：_____红黑树的代码安全性验证_____

项目负责人：__曹钦翔__ 学院（系）：_电子信息与电气工程学院_

指导教师：__曹钦翔__ 学院（系）：_电子信息与电气工程学院_

参与学生：_____秦健行___唐亚周_____

项目执行时间：_2021_年_01_月 至 _2021_年_09_月

红黑树的代码安全性验证

电子信息与电气工程学院 F1903303 唐亚周

指导老师：电子信息与电气工程学院 曹钦翔

摘要

红黑树是一种经典的数据结构，可以实现高效的插入、删除、查找等操作。然而，在红黑树的实现过程中往往会产生由指针带来的错误。因此，验证这种典型数据结构的代码安全性有着现实的意义。本项目在之前研究的基础上，在 Coq 证明助手中使用经验证的软件工具链（Verified Software Toolchain, VST）对于红黑树中查找返回指针的操作进行形式化，并对其正确性进行证明。本项目基于分离逻辑，将红黑树的结点分为存储在结点上的 Address 值，以及内存中 Address 位置对应的 Value 值。本项目试图证明，查找操作返回的指针，其指向的值一定是该结点所对应的值。

关键词：红黑树，Coq 证明助手，VST，形式化验证，分离逻辑

ABSTRACT

The Red-Black tree is a classic data structure that can implement efficient insertion, deletion, and search operations. However, errors caused by pointers often occur in the implementation of Red-Black trees. Therefore, it is of practical significance to verify the code security of this typical data structure. Based on the previous research, this project uses Verified Software Toolchain (VST) in the Coq proof assistant to formalize search operation that returns pointers in the Red-Black tree and prove its correctness. This project is based on separation logic and divides the nodes of the red-black tree into the Address stored on the node and the Value corresponding to the Address location in the memory. This project is trying to prove that the value of the pointer returned by the search operation must be the value corresponding to the node.

KEY WORDS: Red-Black Tree, Coq Proof Assistant, Verified Software Toolchain, Formal Verification, Separation Logic

1. 绪论

红黑树（Red-Black Tree）是一种经典的数据结构，由 Rudolf Bayer 于 1972 年发明。红黑树可以在 O(log n)的时间内完成插入、删除和查找操作，效率较高，因此被广泛应用于 C++标准库等对运行时间有所要求的场景下。

然而，由于红黑树的结构复杂，且在实现过程中常常会用到指针，因此有可能会产生指针错误，在程序中留下隐患。本项目在之前研究的基础上，对红黑树的代码安全性验证做了进一步的研究。

本项目对于红黑树的查找返回指针的操作进行了证明。本文第 2 章介绍了红黑树、Coq 证明助手、VST 证明工具的相关内容，第 3 章在 Coq 中对本项目所证明的红黑树以及相关数据结构进行了形式化，第 4 章尝试使用 VST 对红黑树查找返回指针的操作进行证明，第 5 章提出了本项目得到的主要结论以及对未来的展望。

本课题是在吴姝姝和赵启元两位学长学姐的研究基础上开展的。在之前的研究中，学长学姐们在 Coq 中构造了一棵红黑树，并证明了其插入、删除、查找以及段修改操作的正确性；同

时还使用 VST 对一种特定的红黑树 C 语言实现进行了形式化验证，证明了这段代码所实现的红黑树的插入、删除、查找以及段修改操作的正确性。

本 PRP 小组的另外一名同学秦健行则在之前研究的基础上，实现了红黑树的段统计操作并证明了其正确性。

2. 背景

2.1. 红黑树

红黑树是一种每个结点都带有颜色属性的二叉查找树，其颜色为红色或黑色。除满足二叉查找树的所有要求外，红黑树还有以下性质[1]：

1. 结点是红色或者黑色。
2. 根结点是黑色。
3. 所有叶子结点都是黑色
4. 每个红色结点必须又两个黑色的子结点。
5. 从任一结点到其每个叶子的所有简单路径都包含相同数目的黑色结点。

红黑树的查找操作与普通二叉树相同。红黑树的插入和删除操作有多种实现方式，但总体思路为在寻找插入/删除位置的前后对红黑树的结点进行旋转等操作，使其满足红黑树的性质，从而保证树的平衡性[1]。

2.2. Coq 证明助手

“数学命题 P 的证明是一段书面（或口头）的文本，它对 P 的真实性进行无可辩驳的论证，逐步说服读者或听者使其确信 P 为真。”[2]证明分为形式化证明和非形式化证明。非形式化证明的“读者”一般是人类，也就是用自然语言来让读者理解证明过程。但由于读者身份的多样性，非形式化的证明很难顾及到每一个读者。

而形式化证明的“读者”就是 Coq 这样的程序。Coq 是一个交互式的定理证明辅助工具。它允许用户输入包含数学断言的表达式、机械化地对这些断言执行检查、帮助构造形式化的证明、并从其形式化描述的构造性证明中提取出可验证的（certified）程序。Coq 提供了自动化定理证明的策略（tactics）和不同的决策过程。有许多著名的定理证明工作是基于 Coq 完成的，比如 Georges Gonthier 等人在 2005 年使用 Coq 完成了四色定理的形式化证明[3]。本项目在 Coq 中进行形式化证明。

2.3. VST 证明工具

2.3.1. 霍尔逻辑和分离逻辑

2.3.1.1. 霍尔逻辑

霍尔逻辑（Hoare Logic），又称弗洛伊德-霍尔逻辑（Floyd-Hoare Logic），是由英国计算机科学家 Tony Hoare 开发的形式系统，其用途是使用严格的数理逻辑推理来为计算机程序的正确性提供一组逻辑规则[4]。霍尔逻辑的中心特征是霍尔三元组（Hoare triple），对一段代码的执行如何改变计算的状态进行了描述。霍尔三元组有如下形式：

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

这里的 P 和 Q 是断言，C 是命令。P 被称作前条件，Q 被称作后条件。霍尔三元组在直觉上的含义是，只要 P 在 C 执行前的状态下成立，则 Q 在 C 执行后的状态下成立。霍尔逻辑为简单的指令式编程语言的所有构造提供了公理和推理规则。

2.3.1.2. 分离逻辑

霍尔逻辑在处理指针和内存的相关问题时有一定局限性。分离逻辑（Separation Logic）是霍尔逻辑的一种扩展[5]。分离逻辑的核心内容是分离与（Separating Conjunction）运算，记为$A * B$，表示在两个不相交（disjoint）的内存区域上，一个满足A，另一个满足B。与普通的逻辑与（$A \wedge B$）相比，分离与不仅要求A和B都成立，还要求它们在堆内存上划分出的两个不相交子堆中分别成立。

分离逻辑中还有几个关键的概念：

- emp，代表一块空的堆内存。
- $x \mapsto y$，表示x的值对应的地址，在堆内存上对应的值为y的值。
- $!!R$，表示一个纯命题（pure proposition），即该命题R与内存无关。

2.3.2. VST 简介

经验证的软件工具链（Verified Software Toolchain, VST）是一个用于证明 C 语言程序的功能正确性的工具集，由普林斯顿大学的研究团队开发。

使用 VST 进行形式化验证的步骤如下：首先使用 CompCert（已验证正确性的编译器）的 clightgen 工具从一段 C 语言代码中生成其抽象语法树（Abstract Syntax Tree, AST），然后使用 VST 检验其是否符合用户定义的霍尔三元组。如果符合，则该段 C 语言代码是符合我们的要求的，也就是“正确的”[6]。本项目使用 VST 对红黑树查找返回指针的操作的 C 语言程序进行了形式化证明。

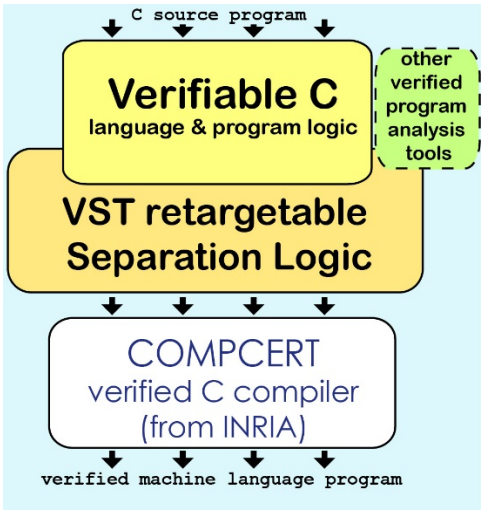

图 1 VST 的工作流程图（来自 VST 项目主页 https://vst.cs.princeton.edu/）

3. 在 Coq 中构造抽象的红黑树并证明

本项目首先在 Coq 中构造出红黑树的数据结构和查找返回指针的函数，并证明其正确性。该部分工作是在之前研究基础上进行的修改，因此下文中着重阐述修改的具体情况。

3.1. 对于指针的处理

由于本项目需要证明红黑树的查找返回指针的操作的正确性，因此在 Coq 中定义的红黑树与其 C 语言实现有所差别。以下分别是 C 语言和 Coq 中对红黑树的定义。

```
struct tree {
    int color;
    int key;
    unsigned int value;
    struct tree *left, *right, *par;
};
```

```
Inductive RBtree : Type :=
| Empty :  RBtree
| T :  color -> RBtree -> Key -> Address -> RBtree -> RBtree.
```

本项目将红黑树的结构一分为二：结点上存储着$Address$值，而在内存上$Address$所对应的位置则存储着$Value$值，如图 2 所示。因此，在 Coq 定义的$RBtree$中，每个结点存储的值不再是$Value$，而是这个$Value$所在的$Address$。本项目试图证明，在内存上，查找操作返回的这个$Address$一定存储着该结点对应的$Value$。

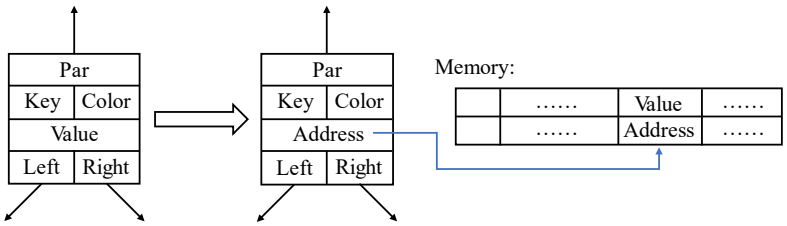

图 2 修改后的红黑树结构

3.2. 查找返回指针操作与段修改操作的冲突

以下是 C 语言中查找返回指针操作的具体实现。

```
unsigned int* lookup_pointer(struct tree*p, int x){
    for(; ; ) {
        if(p == NULL){
            return NULL;
        }
        if(p->key < x){
            p = p->right;
        }else if(p->key > x){
            p = p->left;
        } else{
            return &(p->value);
        }
    }
}
```

在程序中，如果调用$lookup_pointer$函数获得了该结点所对应的$Value$的地址，那么就可以直接获得$Value$的值。但之前研究中所实现的段修改操作，是通过在子树的根结点上增加一个tag变量，并在向下查询时获取tag的值与目标结点所对应的$Value$相加来实现的。因此，该操作与段修改操作有冲突。所以本项目去掉了对段修改操作的实现。

4. 使用 VST 进行证明

4.1. VST 中的类型转换

在 VST 中，C 变量的类型被定义为val，可以表示整型数、浮点型数或地址[6]。函数$Vint$可以将 Coq 中 32 位的整型数（int类型）转化为 C 中的val类型，函数$Int.repr$可以将 Coq 中的整型数（Z类型）转化为 32 位的整型数（int类型）。

```
Inductive val: Type :=
| Vundef: val
| Vint: int -> val
| Vlong: int64 -> val
| Vfloat: float -> val
| Vsingle: float32 -> val
| Vptr: block -> ptrofs -> val.
```

4.2. 红黑树的形式化

本项目所构造的红黑树，其结点上存储着$Address$值，而在内存上$Address$所对应的位置则存储着$Value$值。这相当于把$RBtree$的$Value$这个数据成员从整个结构体中拿出来，并在内存中对其进行声明。这样就借助分离逻辑避免了同一片内存区域被多个指针所指向的问题。

红黑树在 VST 中的定义$rbtree_rep_pointer$如下。由于本项目要把$Address$部分单独拿出来考虑，因此不能使用$data_at$来表示红黑树在内存中的存储情况，而要对每个数据成员使用$field_at$，再使用分离与连接词将其连接。而对于$Address$，则需要使用$field_address$将其与$Value$建立联系。对于任意满足 3.1 中定义的$RBtree$结构的t，其要么是空树，要么满足以下结构：

$$T\ col\ lch\ key\_\ address\_\ rch$$

其中，col、lch、$key\_$、$address\_$、rch表示t的各个参数，它们都和指针p所指向内存区域的不相交划分有所对应。p所对应的地址上存储着t_struct_rbtree类型的对象。t_struct_rbtree是由 C 代码生成的语法树中定义而来，也就是说，该对象与 C 代码中定义的$struct\ tree$有着相同的数据成员。

```
Definition t_struct_rbtree := Tstruct _tree noattr.
```

具体的对应关系是：

- 该对象的$_color$、$_key$数据成员的位置分别存储着经过类型转换后的col、key。
- 该对象的$_left$、$_right$数据成员的位置分别存储着p_lch、p_rch，而p_lch、p_rch代表着指向左右子树的指针，分别指向lch、rch，通过 $rbtree_rep_pointer\ lch\ p_lch\ p$和 $rbtree_rep_pointer\ rch\ p_rch\ p$进行了递归的定义。
- 该对象的$_par$数据成员的位置存储着p_par，代表着指向父的指针。
- 而该对象的$_value$数据成员没有在p所对应的地址上使用$field_at$来进行描述。它的信息使用$field_address$以纯命题的方式进行定义：其位置所表示的地址，就是$address_$的值。

```
Fixpoint rbtree_rep_pointer (t: RBtree) (p: val) (p_par: val) : mpred :=
 match t with
 | T col lch key_ address_ rch =>
   !! (Int.min_signed <= key_ <= Int.max_signed /\
     is_pointer_or_null p_par) &&
   !! (address_ = field_address t_struct_rbtree [StructField _value] p) &&
   EX p_lch : val, EX p_rch : val,
   field_at Tsh t_struct_rbtree [StructField _color] (Vint (Int.repr (Col2Z col))) p
   * field_at Tsh t_struct_rbtree [StructField _key] (Vint (Int.repr key_)) p
   * field_at Tsh t_struct_rbtree [StructField _left] (p_lch) p
   * field_at Tsh t_struct_rbtree [StructField _right] (p_rch) p
   * field_at Tsh t_struct_rbtree [StructField _par] (p_par) p
   * rbtree_rep_pointer lch p_lch p * rbtree_rep_pointer rch p_rch p
 | E => !! (p = nullval /\ is_pointer_or_null p_par) && emp
 end.
```

在$field_at$相关的命题和$rbtree_rep_pointer$的递归定义命题之间，使用了分离与连接词（$*$），表示每个命题所涉及的内存区域互不相交，避免了多个指针指向同一片内存区域引起的问题。

4.3. 其它辅助数据结构的形式化

查找操作是从根结点向叶子结点不断向下的。要证明这个循环过程的正确性，本项目需要定义一个循环不变量（$loop\ invariant$），也就是每次循环都满足的性质。因此在证明中，本项目会用到一些辅助的数据结构，$half\ tree$和$partial\ tree$。$half\ tree$表示有左子树或右子树为空的树，其在 Coq 中的定义如下。

```
Definition Half_tree : Type := (bool * color * Key * Address * RBtree ).
```

其中的$bool$类型代表空缺的位置在其左子树（为$true$）还是右子树（为$false$）。而$partial_tree$则是由$half_tree$构成的$list$，也就表示整棵树中有一个空缺的位置。

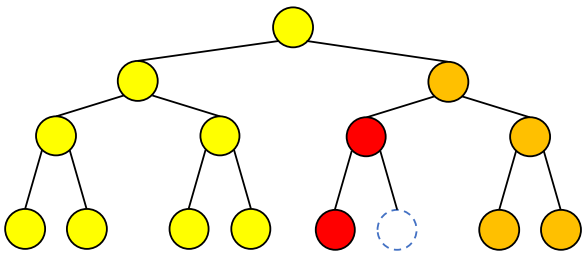

图 3 $partial\ tree$和$half\ tree$示意图

如图 3 所示，蓝色虚线结点为空缺位置，红色结点、橙色结点和黄色结点代表 3 个连续的$half\ tree$，它们共同构成了一个$partial\ tree$。

与 4.1 中原理相同，在 VST 中定义$partial_tree_rep_pointer$时也不能使用$data_at$，而要使用$field_at$和$field_address$相结合的方法。对于任意$partial_tree$（$list\ Half\_tree$），其要么为空，要么满足以下结构：

$$(va, c, k, addr, sib) :: l$$

这里用到了 6 个指针：指针p指向该空缺位置，p_par指向$partial\ tree$最低的一个$half\ tree$的根结点（也就是空缺位置的父结点），p_gpar指向空缺位置的祖父结点，p_sib指向空缺位置的兄弟结点sib，p_root指向整棵树的根结点，p_top指向$partial\ tree$最高的一个$half\ tree$的根结点。我们主要关注p_par所指向的内存区域，该内存区域存储着一个t_struct_rbtree类型的对象。

- 该对象的$_color$、$_key$数据成员的位置分别存储着经过类型转换后的c、k。
- 该对象的$_left$、$_right$数据成员的位置存储着p和p_sib，但具体的对应关系由va的值来确定。p_sib指向sib，通过$rbtree_rep_pointer\ sib\ p_sib\ p_par$进行了定义。
- 该对象的$_par$数据成员的位置存储着p_gpar，通过 $partial_tree_rep_pointer\ l\ p_root\ p_par\ p_gpar\ p_top$进行了递归的定义。
- 而该对象的$_value$数据成员同样没有在p_par所对应的地址上使用$field_at$来进行描述。它的信息使用$field_address$以纯命题的方式进行定义：其位置所表示的地址，就是$addr$的值。

```
Fixpoint partial_tree_rep_pointer (t : list Half_tree) (p_root p p_par p_top : val) : mpred :=
 match t with
 | [] => !! (p = p_root /\ p_par = p_top) && emp
 | (va, c, k, addr, sib) :: l  =>
   EX p_gpar: val, EX p_sib : val,
       !! (Int.min_signed <= k <= Int.max_signed) &&
       !! (addr = field_address t_struct_rbtree [StructField _value] p_par) &&
       rbtree_rep_pointer sib p_sib p_par *
       partial_tree_rep_pointer l p_root p_par p_gpar p_top *
       field_at Tsh t_struct_rbtree [StructField _color] (Vint (Int.repr (Col2Z c))) p_par
       * field_at Tsh t_struct_rbtree [StructField _key] (Vint (Int.repr k)) p_par
       * field_at Tsh t_struct_rbtree [StructField _left] (if va then p_sib else p) p_par
       * field_at Tsh t_struct_rbtree [StructField _right] (if va then p else p_sib) p_par
       * field_at Tsh t_struct_rbtree [StructField _par] (p_gpar) p_par
 end.
```

在各个涉及内存的命题之间，使用了分离与连接词（$*$），表示每个命题所涉及的内存区域互不相交，避免了多个指针指向同一片内存区域引起的问题。

4.4. 对 lookup 的证明

4.4.1. lookup_pointer 函数的形式化

首先定义一个函数，把在 Coq 中定义的$lookup$的返回值转化为 VST 中的 val 类型。

```
Definition Lookup2val (x: Key) (t: RBtree) : val :=
 match lookup x t with
 | None    => nullval    (* Default value *)
 | Some v  => v
 end.
```

然后定义$lookup_pointer$应该满足的霍尔三元组。PRE 中描述的是其前条件，POST 中描述了其后条件，而 SEP 中描述的则是分离逻辑相关的性质。

```
Definition lookup_pointer_spec :=
 DECLARE _lookup_pointer
  WITH x: Key, p: val, t: RBtree, p_par : val
  PRE  [ tptr t_struct_rbtree, tint ]
    PROP (Int.min_signed <= x <= Int.max_signed;
      is_pointer_or_null p_par)
    PARAMS (p; Vint (Int.repr x))
    SEP (rbtree_rep_pointer t p p_par)
  POST [ tptr tuint ]
    PROP ()
    RETURN ((Lookup2val x t))
    SEP (rbtree_rep_pointer t p p_par).
```

4.4.2. 引理的定义和证明

在正式开始证明之前，本项目需要证明几个引理。由于篇幅有限，本文只阐述其定义而省略了证明部分。

首先是通常在定义了新的*mpred*类型（比如本项目的*rbtree_rep_pointer*和*partial_tree_rep_pointer*）后需要证明的*saturate_local*和*valid_pointer*引理，用于扩展 VST 的提示数据库（Hint Database）。前者用于从空间事实中提取纯命题事实的引理，后者用于从空间引理中提取有效指针事实的引理。[7]

然后是用于解决空指针的*rbtree_rep_nullval*引理。

```
Lemma rbtree_rep_nullval : forall (t: RBtree) p_par,
  rbtree_rep_pointer t nullval p_par |-
- !! (t = Empty) && !! (is_pointer_or_null p_par) && emp.
Proof.
```

最后是用于解决子树和*partial tree*的*reconstruction_lemma*引理。

```
Lemma reconstruction_lemma :
  forall ls t root p p_par p_top,
  !! (is_pointer_or_null p_top) &&
    rbtree_rep_pointer t p p_par * partial_tree_rep_pointer ls root p p_par p_top
  |-- rbtree_rep_pointer (complete_tree ls t) root p_top.
```

4.4.3. *lookup_pointer*的证明

最后就是*lookup_pointer*函数的证明，由于篇幅有限，本文只进行简单阐述。

```
Theorem body_lookup_pointer: semax_body Vprog Gprog f_lookup_pointer lookup_pointer_spec.
```

证明过程中最重要的就是对于循环不变量的定义。本项目定义的循环不变量如下：在任意一次循环结束后，都存在*RBtree*类型的变量t'，*Half_tree*类型的列表ls'（*partial tree*），使得在原来的树t中查找x与在t'中查找x的结果相同，且t'与ls'结合起来就是t。同时t'和ls'也满足*rbtree_rep_pointer*和*partial_tree_rep_pointer*在内存上的分离逻辑命题。

```
forward_loop
(EX t' : RBtree, EX ls' : list Half_tree,
 EX p' : val, EX p_par' : val,
 PROP (is_pointer_or_null p_par';
  lookup x t = lookup x t';
  complete_tree ls' t' = t)
 LOCAL (temp _p p'; temp _x (Vint (Int.repr x)))
 SEP (rbtree_rep_pointer t' p' p_par';
  partial_tree_rep_pointer ls' p p' p_par' p_par))%assert.
```

5. 结论与展望

5.1. 本项目的结论

本项目对红黑树的代码安全性验证进行了研究。本项目在之前研究的基础上，选取了红黑树的查找返回指针的操作进行了研究。本项目首先在 Coq 证明助手中构造了抽象的红黑树以及查找返回指针的操作，并证明了其正确性。然后在 VST 中应用分离逻辑形式化地定义了红黑树及其辅助数据结构，并完成了查找返回指针的操作的证明。

本项目将红黑树的结点一分为二，将结点上存储的$Value$值去掉，改为存储$Address$值，而在内存上$Address$所对应的位置则存储着$Value$值。这样避免了多个指针指向存储$Value$的内存区域引起的问题。

本项目是应用 VST 证明数据结构和算法正确性的一次实践，为之后的研究提供了参考。

5.2. 对未来的展望

针对本项目的不足之处，在未来有以下可探索的完善与改进方向

- 本项目的查找返回指针操作与之前研究的段修改操作有冲突，可以探索通过改进实现方式实现两者的兼容。
- 本项目证明了查找返回指针操作的正确性，可以进一步证明指针相关的其它操作，比如$prev$、$next$等操作的正确性。
- 本项目以及之前的研究基础，都关注于普通红黑树的形式化验证。在未来也可以将其拓展到红黑树的变体，比如 Robert Sedgewick 提出的左倾红黑树[8]、近年来有多位学者研究的并发红黑树[9-10]等。

参考文献

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009

[2] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey, Software Foundations, Volume 1, Version 6.1, https://softwarefoundations.cis.upenn.edu/lf-current/index.html, 2021.

[3] Gonthier G. Formal proof–the four-color theorem[J]. Notices of the AMS, 2008, 55(11): 1382-1393.

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

[5] O'Hearn P. Separation logic[J]. Communications of the ACM, 2019, 62(2): 86-95.

[6] Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds, Verifiable C, Version 2.5, https://github.com/PrincetonUniversity/VST/blob/master/doc/VC.pdf, 2021.

[7] Andrew W. Appel, Lennart Beringer, and Qinxiang Cao, Software Foundations, Volume 5, Version 1.1.1, https://softwarefoundations.cis.upenn.edu/vc-current/index.html, 2021.

[8] Sedgewick R. Left-leaning red-black trees[C]//Dagstuhl Workshop on Data Structures. 2008, 17.

[9] Natarajan A, Savoie L H, Mittal N. Concurrent wait-free red black trees[C]//Symposium on Self-Stabilizing Systems. Springer, Cham, 2013: 45-60.

[10] Besa J, Eterovic Y. A concurrent red–black tree[J]. Journal of Parallel and Distributed Computing, 2013, 73(4): 434-449.

致谢

感谢曹钦翔老师，让我们学会了 Coq 证明助手和 VST 的使用，每周的组会上与我们进行讨论，为我们指引了正确的方向，带领我们初步接触了形式化验证的科研。感谢吴姝姝学姐和赵启元学长，在我们遇到困难时帮助我们，同时他们的研究工作也是本项目的基础。感谢 PRP 项目成员秦健行，以及实验室的另外三位成员董文韬、王崇华、毛力源，与他们的讨论让我的思路更加清晰。最后感谢上海交通大学的 PRP 计划，让我们能有机会接触科研。