红黑树的功能扩展与形式化验证

上海交通大学第 39 期 PRP 项目答辩

秦健行，唐亚周[1]
指导老师：曹钦翔

[1] 电子信息与电气工程学院
上海交通大学

2021 年 10 月 13 日

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

总览

红黑树介绍

红黑树

- 一种经典的数据结构，可以存储定义了序关系的对象集合
- 支持稳定高效的查询与修改等操作

红黑树的应用

- Linux 内核使用红黑树作为数据管理结构
- 许多程序语言的标准库使用红黑树实现集合与关联表结构

现有的红黑树验证工作与广泛使用的指令式，特别是直接操作内存的底层实现具有较大差距。

霍尔逻辑和分离逻辑

- **霍尔逻辑**（Hoare Logic）：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。
  - **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。

$$P\{C\}Q$$

P（前条件）和 Q（后条件）是断言，C 是命令。
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。
  - 霍尔逻辑的局限性：处理指针和内存的相关问题。
- **分离逻辑**（Separation Logic）是霍尔逻辑的一种扩展。
  分离与（Separating Conjunction）**运算**：$A * B$
  - 在两个不相交的内存区域上，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。

关联表抽象

为了证明红黑树操作的正确性，我们定义红黑树与关联表的抽象对应关系，并证明这种关系在操作前后的保持性。

```
Definition relate_map := K -> option V.
Inductive Abs : rbtree -> relate_map -> Prop := (* omitted *).
```

例如改变树本身的操作的正确性可以被形式化为:

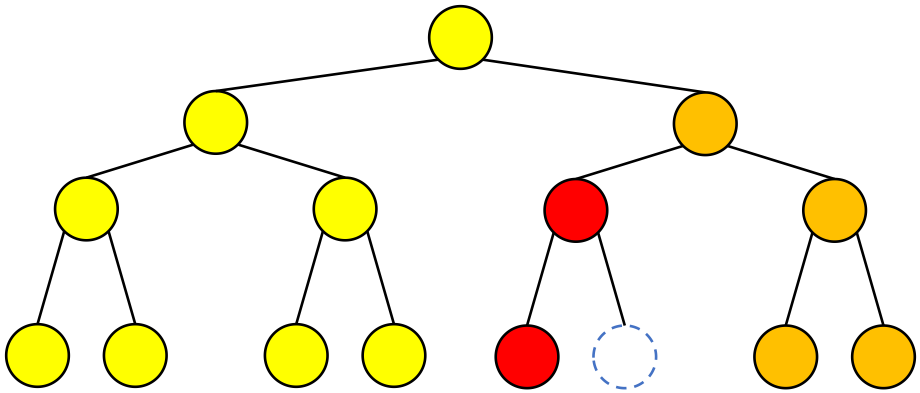
```
Definition operation_abs
  (a: Type)
  (opt: rbtree -> a -> rbtree)
  (opm: relate_map -> a -> relate_map) :=
  forall t m x, Abs t m -> Abs (opt t x) (opm m x).
```

用半树表达操作路径

在 C 语言中，红黑树数据结构的实现通常要使用指向父节点的指针以回溯操作路径。这种环形引用在 Coq 中是无法直接表达的。
半树（*half_tree*）是由一棵完整的红黑树去除其中一个子树所形成的结构，并添加一个布尔类型信息指示去除的子树是原本的左子树还是右子树。

```
Definition half_tree : Type := bool * color * K * V * rbtree.
```

这样就可以利用半树组成的列表（称作 *partial_tree*）来表示操作路径。



段修改

为了高效地在红黑树上执行段修改，可以使用延迟计算的技巧。
在红黑树的每个节点上存储额外的标记信息用于表示该节点对应的子树上所有的键值对都应该进行对应标记的更新。

```
Fixpoint change_segment
  (lo hi: K) (delta: tag) (s: rbtree) (segl segh: K): rbtree :=
if (hi <? lo) then s else (
if (hi <? segl) || (segh <? lo) then s else (
if (lo <=? segl) && (segh <=? hi) then tag_tree delta s else
  match s with
  | RbE => RbE
  | RbT l (co, k, v, t) r =>
    (* 修改两个子树的标记，且如果当前节点在区间内，则对值应用标记 *)
  end)).
```

部分的实际计算过程被推迟到查询操作经过对应节点时发生，此时节点上的标记被应用到该节点的值并被下推一层到两个子树。

段统计

在红黑树上支持高效的段统计操作，一个方法是在红黑树的每个节点内额外存储一个统计量，表示该子树对应键值集合的统计量。

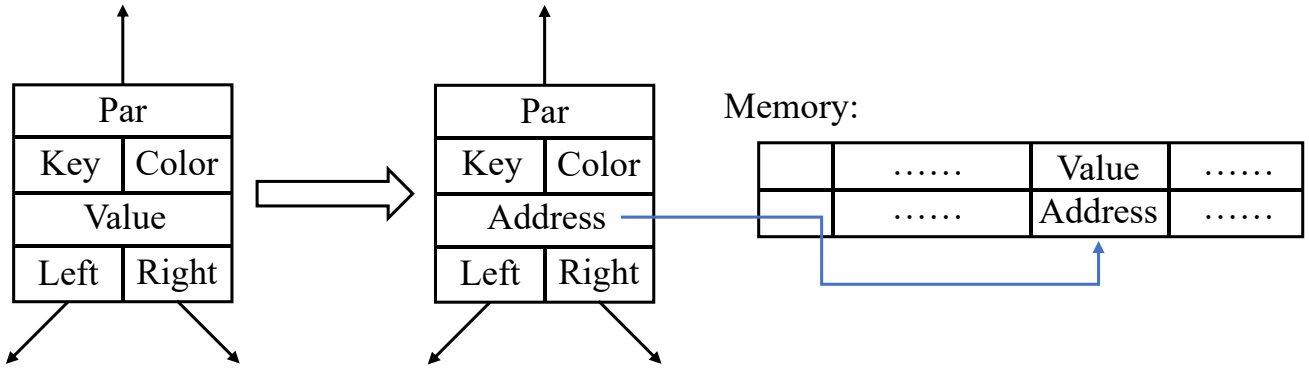
```
Fixpoint stat_segment (lo hi: K) (s: rbtree) (segl segh: K): stat :=
if (hi <? lo) then default_stat else (
if (hi <? segl) || (segh <? lo) then default_stat else (
if (lo <=? segl) && (segh <=? hi) then get_stat s else
  match s with
  | RbE => default_stat
  | RbT l (co, k, v, st) r =>
    (* 对左右子树分别统计并合并 *)
  end)).
```

这样减少了对同一段数据的重复统计。

Coq 中的红黑树定义

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

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。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。



与段修改操作的冲突

- 调用 *lookup_pointer* 函数获得了该结点所对应的 *Value* 的地址 $\Rightarrow$ 直接获得 *Value* 的值
- 段修改操作：延迟修改

该操作与段修改操作有冲突 $\Rightarrow$ 这一部分去掉了对段修改操作的实现

红黑树的形式化

- *_value* 的信息使用 *field_address* 以纯命题进行定义：其位置所表示的地址就是 *address_* 的值。
- 多个命题之间使用分离与（$*$）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。

```
Fixpoint rbtree_rep_pointer (t: RBtree) (p: val) (p_par: val) : mpred :=
match t with
| E => !! (p = nullval /\ is_pointer_or_null p_par) && emp
| 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
end.
```

红黑树的形式化

- *_value* 的信息使用 *field_address* 以纯命题进行定义：其位置所表示的地址就是 *address_* 的值。
- 多个命题之间使用分离与（$*$）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。

```
Fixpoint rbtree_rep_pointer (t: RBtree) (p: val) (p_par: val) : mpred :=
match t with
| E => !! (p = nullval /\ is_pointer_or_null p_par) && emp
| 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
end.
```

partial_tree 的形式化

```
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.
```

partial_tree 的形式化

```
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.
```

lookup_pointer 的证明

其 Specification 以霍尔三元组的形式定义如下（有省略）：

$$\{SEP(rbtree\_rep\_pointer\ t\ p\ p\_par)\}$$
$$\_lookup\_pointer$$
$$\{RETURN(Lookup2val\ x\ t); SEP(rbtree\_rep\_pointer\ t\ p\ p\_par)\}$$

- p 与 x 是 C 语言中函数的参数；
- *Lookup2val* 使用了 Coq 中定义的 *lookup*（已证明正确性）；
- *SEP* 中是分离逻辑相关的性质。

结论

- 本项目对红黑树的代码安全性验证进行了研究，在之前研究的证明框架基础上进行了扩展和证明。
- 本项目给出了基于段操作的扩展功能实现，并利用 VST 工具对对应 C 程序实现进行了初步的证明。
- 本项目在 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。
- 本项目是应用 VST 证明数据结构和算法正确性的一次实践，展现了程序形式化证明的应用前景，为之后的研究提供了参考。

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。
- 本项目可以尝试在目前的基础上支持更为通用的扩展功能。如 Linux 内核实现中红黑树的传播回调、复制回调与树旋转回调等。
- 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，比如左倾红黑树、并发红黑树等。