A Pointer Logic Dealing with Uncertain Equality of Pointers

Hongjin Liang*, Yu Zhang†, Yiyun Chen†, Zhaopeng Li†, Baojian Hua‡
*Special Class for the Gifted Young, University of Science and Technology of China
†Department of Computer Science and Technology, University of Science and Technology of China
‡School of Software Engineering, University of Science and Technology of China
Hefei, 230026, China
{yiyun, yuzhang}@ustc.edu.cn, {lhj1018, zpli, huabj}@mail.ustc.edu.cn

Abstract

We have designed a pointer logic for a C-like programming language - PointerC. The pointer logic is an extension of Hoare logic, and it uses the idea of precise alias analysis in pointer program verification to support safety verification of programs in which equality of pointers is well-regulated. In this work, we present an extension to the pointer logic by introducing a set of uncertain-equality pointer access path sets, so that we can reason in the extended pointer logic about properties of programs which manipulate data structures like directed graph in which equality of pointers is uncertain.

1. Introduction

With the increasingly widespread use of software, the safety of software is becoming crucial. Formal program verification technique provides a way to reason about the safety of programs. Hoare logic is widely used in verification of imperative programs, but it's difficult to reason in Hoare logic about pointer programs with complex aliases. The major difficulty is in the treatment of variable assignment where the substitution affects only relevant variable.

Separation logic [1], which is an extension of Hoare logic, aims to simplify reasoning about programs with shared mutable data structures, *i.e.*, structures where an updatable field can be referenced from more than one point. In separation logic, assertions `P` and `Q` in separation conjunction `P*Q` hold for disjoint portions of the addressable storage. Separation logic is widely used in assembly language level program verification. However, in separation logic, the disjoint heaps are deemed to have no effect on each other but there is no mechanism to describe connections between them. Thus it is difficult to get intuitive impression about the shape of the structure like binary tree, DAG and circular linked list without other support such as S-expression.

To study pointer program verification, we have designed the PointerC language [2] which is a C-like language with dynamic memory allocation and deallocation. The elementary safety policy is that there are no operations such as dereference and free on null pointers or dangling pointers, no use of dangling pointers as parameter of function call, and no memory leaks during the program execution. To reason about such properties of the program, we adopt a method combining techniques of type and logic systems. In order to design a simple yet sound type system, we introduce side conditions in the typing rules which make restrictions on the value of the syntactic expressions. To check these side conditions, a pointer logic [3, 4, 6] has been designed for PointerC. The pointer logic is an extension of Hoare logic. It is used to deduce the precise pointer information at each program point such as whether a pointer is null, dangling, or valid (a valid pointer points to an object in the heap) and the equality between valid pointers. All information is used to prove whether pointer programs satisfy the side conditions, thus it can support safety verification of pointer programs and verification of other properties. Afterwards, we have extended the pointer logic with limited pointer arithmetic to support safety verification of operations on dynamic arrays [7]. Furthermore, we have implemented a certifying compiler prototype plcc [4, 5] which can generate proof-carrying assembly codes.

However, the original pointer logic could only deal with data structures, in which equality of pointers is well-regulated, such as linked list and binary tree. Structures like DAG (directed acyclic graph) and other graph, in which equality of pointers is uncertain, are unsupported. On the other hand, the pointer logic needs excessively precise information about equality of pointers when reasoning about properties of programs, so that it's necessary to moderately relax the restrictions to allow partly uncertain information. As a result, a set of uncertain-equality access path sets is introduced. Pointers in such an access path set are equal (that is, they point to the same object in the heap), but they may also be equal to pointers in another such access path set. One of the problems we encounter is that memory leak may exist when the only pointer in such an access path set is assigned. Another problem is that when we free an object pointed to by a pointer in such a set, we can't ensure that all the pointers point to the object are labeled as dangling pointers. If these situations do not occur, the original pointer logic

is appropriate. We extend the pointer logic so that if these situations occur, then we give warnings or even break off the verification; otherwise we still use the original approach.

This paper extends the pointer logic based on the idea above. The main contributions are:

1) We introduce a set of uncertain-equality access path sets into the original access path sets and *pointer non-membership assertions* into the assertion language. Thus the pointer logic can deal with data structures in which equality of pointers is uncertain.
2) We apply the pointer logic to verification of practical programs with non-single structures for the first time. The data structures used in practical programs are often complicated that consist of several typical data structures such as singly-linked list and doubly-linked list. Applying the pointer logic to verification of such programs may show the potential and practicability of the pointer logic.

The rest of the paper is organized as follows. Section 2 extends the pointer logic to deal with uncertain equality of pointers. Section 3 gives two examples to illustrate how pointer logic is used for verification. Section 4 presents the related work and Section 5 concludes our work.

2. Design of Pointer Logic Dealing with Uncertain Equality of Pointers

To deal with uncertain equality of pointers in the pointer logic, we have to introduce a set of uncertain-equality access path sets, and extend the assertion language, specifications and inference rules.

2.1. Introduction to PointerC and Pointer Logic

PointerC is a C-like programming language in which the pointer type is emphasized. In PointerC, pointer variables can only be used in assignment statements, equality test expressions, in operations like storing and loading the value which they are pointing to, and as parameters of functions. Pointer arithmetic and the address-of operator (&) are forbidden. Functions `malloc` and `free` are regarded as predefined functions in PointerC which satisfy the elementary safety policy. In addition, short-circuit calculation is not adopted in evaluation of PointerC boolean expressions so that PointerC boolean expressions can be used directly in assertions.

In the original pointer logic [6] designed for PointerC, the basic idea is to represent memory states by means of sets of pointers. We have designed three kinds of pointer access path sets: a $\mathcal{N}$ set for null pointer set, a $\mathcal{D}$ set for dangling pointer set, and a set of Π sets for valid pointers. Pointers in one set of Π are pointing to the same object in the heap (that is, they have the same *rvalue*) and pointers in different sets of Π are pointing to different objects.

The access path sets in the pointer logic can be used in inductive auxiliary definitions of data structures. For example, binary tree can be defined as follows:

$$\mathsf{tree}(\texttt{s}) \triangleq \{\texttt{s}\}_{\mathcal{N}} \vee (\{\texttt{s}\} \wedge \mathsf{tree}(\texttt{s->l}) \wedge \mathsf{tree}(\texttt{s->r}))$$

in which `s` is a pointer pointing to:

```
struct BTNode {struct BTNode *r, *l;}
```

$\{\texttt{s}\}_{\mathcal{N}}$ stands for an empty tree and $\{\texttt{s}\} \wedge \mathsf{tree}(\texttt{s->l}) \wedge \mathsf{tree}(\texttt{s->r})$ is used to denote a non-empty tree. By using valid pointer set, it is convenient to express all of the valid pointers in the tree are not equal to each other. Because when expanding all references of the inductive definitions, all valid pointers appear in different subsets of Π.

2.2. Uncertain-equality Access Path Sets

In structures like graphs, there are no rules indicating whether pointers are equal, *i.e.*, there exists valid pointers of which equality is uncertain. Such data structures are unsupported in the original pointer logic. Thus, we introduce a set of uncertain-equality pointer access path sets based on the original pointer logic and use $\mathcal{U}$ to denote it. On the semantic model of PointerC, pointers in one set of $\mathcal{U}$ are pointing to the same object in the heap, and distinguished from pointers in sets of Π, pointers in different sets of $\mathcal{U}$ may be pointing to the same object. Pointers in Π and $\mathcal{U}$ are all called valid pointers. Ψ is used to denote $\mathcal{N}$, $\mathcal{D}$, Π and $\mathcal{U}$ in the rest of the paper. Note that although we use the name "uncertain-equality access path sets", the uncertainty only exists between such sets. Pointers in each set are definitely equal. This approach can be interpreted as a relaxation of the original restrictions on sets like Π.

Since not any Ψ, that is formed by arbitrary partitions of pointer access path sets, can express the mentioned meaning, we need to define *legal* Ψ to describe data structures. As $\mathcal{U}$ is introduced, we need to think on the basis of the original definition of legal Ψ [6] about the characteristics of access paths in $\mathcal{U}$, such as pointers in $\mathcal{U}$ are prefixed by pointers in Π or $\mathcal{U}$, and the relationships between sets of $\mathcal{U}$ and other sets, such as the prefix of an access path in Π is not in $\mathcal{U}$.

Firstly extend the definition of alias of access path to deal with $\mathcal{U}$. Result access paths by adding same suffix to pointers in a same set of Π or $\mathcal{U}$ are aliases.

Extended legal Ψ should satisfy the following conditions:

1) All declared pointer variables must appear in Ψ.
2) For each pointer `p` of Π, if it points to a structure with a pointer field named `r`, then some alias of `p->r` is in Ψ.
3) Any two different pointers in Ψ are not aliases.
4) Every prefix of each pointer in Ψ has an alias in Π or $\mathcal{U}$.
5) For each pointer `p` of $\mathcal{U}$, if it points to a structure with a pointer field named `r`, then some alias of `p->r` is in $\mathcal{U}$, $\mathcal{N}$ or $\mathcal{D}$.

$$
\begin{array}{rcl}
assertion & ::= & boolexp \\
& | & \neg assertion \mid assertion \vee assertion \\
& | & assertion \wedge assertion \mid (assertion) \\
& | & \forall ident : domain.assertion \\
& | & \exists ident : domain.assertion \\
& | & \{lvalset\} \mid \{lvalset\}_{\mathcal{N}} \mid \{lvalset\}_{\mathcal{D}} \\
& | & [lvalset] \mid ident(lval) \\
& | & lval \notin [lvalset] \mid lval \notin ident(lval) \\
domain & ::= & \mathbf{N} \mid exp..exp \\
lvalset & ::= & lvalset, lval \mid lval \\
lval & ::= & id \mid lval\texttt{->}id \mid lval(\texttt{->}ident)^{exp}
\end{array}
$$

Figure 1. The Assertion Language

2.3. Extension of Assertion Language

We extend the assertion language of the original pointer logic [6] to support $\mathcal{U}$. Moreover, to describe non-equality between pointers in different sets of $\mathcal{U}$, we need to introduce non-membership assertions that present a pointer in some set of $\mathcal{U}$ does not belong to another set of $\mathcal{U}$, in addition to the boolean expressions denoting equality or non-equality of two pointers.

The syntax of the extended assertion language is given in Figure 1. Pay attention to the difference between $\{lvalset\}$ for an access path set of Π and newly added $[lvalset]$ for an access path set of $\mathcal{U}$. $\{lvalset\}_{\mathcal{N}}$ and $\{lvalset\}_{\mathcal{D}}$ are still used to stand for access path set of `NULL` pointers and dangling pointers respectively. The newly added assertions $lval \notin ident(lval)$ and $lval \notin [lvalset]$ are *pointer non-membership assertions*.

Intuitively, pointer non-membership assertion is a restriction to $\mathcal{U}$ which can reduce the uncertainty in equality of pointers. It is a bridge connecting $\mathcal{U}$ and Π. For example, binary DAG can be defined as:

$$\texttt{dag(p)} \triangleq \{\texttt{p}\}_{\mathcal{N}} \vee (\texttt{[p]} \wedge \texttt{dag(p->l)} \wedge \texttt{dag(p->r)} \wedge \texttt{p} \notin \texttt{dag(p->l)} \wedge \texttt{p} \notin \texttt{dag(p->l)})$$

By using uncertain-equality access path set and pointer non-membership assertion, it is convenient to express a valid pointer in the graph may be equal to others but will not be equal to any pointers in the left or right subgraph. $\texttt{p} \notin \texttt{dag(p->l)}$ indicates that `p` is not a member of any pointer sets which expanded from the inductive definition of `dag(p->l)`. When `dag(p->l)` is not an empty graph, there is:

$$\texttt{p} \notin \texttt{dag(p->l)} = \texttt{p} \notin \texttt{[p->l]} \wedge \texttt{p} \notin \texttt{dag(p->l->l)} \wedge \texttt{p} \notin \texttt{dag(p->l->r)}$$

Logic negation operation ($\neg$) cannot be applied to pointer non-membership assertions, because the commonly used definitions of data structures do not have such demand.

2.4. Extension of Assertion Calculus

In the original pointer logic [6], some equivalence and implication axioms for Ψ are designed, including 1) Equivalence axioms of $\mathcal{N}$ and $\mathcal{D}$: two $\mathcal{N}/\mathcal{D}$ sets are equivalent with the combined $\mathcal{N}/\mathcal{D}$; 2) Axioms of illegal Ψ and assertions: assertions containing illegal Ψ are false; 3) Equivalence axioms of access path set and assertion: two legal Ψ are equivalent if the corresponding access path sets are equivalent; 4) Axiom schemas of boolean expressions and Ψ: if the boolean expression such as `p==NULL`, `p!=NULL`, `p==q` and `p!=q` (`p` and `q` are all pointers) added in the assertion is consistent with Ψ then it is absorbed; otherwise the whole assertion, including the boolean expression and Ψ, implies false.

After introducing $\mathcal{U}$ and pointer non-membership assertions, these axioms should be extended. Take part 4) as an example:

4) Axiom schemas of boolean expressions and Ψ
   Assume Π consists of n access path sets $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and $\mathcal{U}$ consists of m access path sets $\mathcal{T}_1, \ldots, \mathcal{T}_m$. Q is an assertion not including access path sets. $\mathcal{S}^{\texttt{p}}$ is short for a pointer set which includes an alias of pointer `p`. $\mathcal{S}^{\texttt{p,q}}, \mathcal{T}^{\texttt{p}}$ and $\mathcal{T}^{\texttt{p,q}}$ have similar meanings. Note that in this part, $\mathcal{U}$ and Π are on equal terms so that the original axioms for Π could be almost copied as axioms for $\mathcal{U}$. Due to the limited space, such axioms are omitted. In addition, two axioms are newly added when `p` and `q` are repectively members of some set of Π and $\mathcal{U}$:

   $$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-1} \wedge \mathcal{T}^q \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q \wedge \texttt{p==q} \Longrightarrow \text{false}$$

   $$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-1} \wedge \mathcal{T}^q \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q \wedge \texttt{p!=q} \Longrightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \, \mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-1} \wedge \mathcal{T}^q \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q$$

Other axioms are introduced because of the particularity of $\mathcal{U}$ and pointer non-membership assertions. When `p` and `q` are in different sets of $\mathcal{U}$, the assertion `p==q` will make the two sets of $\mathcal{U}$ combined, and then go on to combine the sets which includes access paths prefixed by `p` and `q`. On the other hand, the assertion `p!=q` ensures that the pointers in the two sets of $\mathcal{U}$ are definitely unequal, thus it can be translated into pointer non-membership assertion. Besides, special equivalence and implication axioms for pointer non-membership assertion should be designed. These newly added axioms are given as follows:

5) Axioms of combination of $\mathcal{U}$
   When the assertion `p==q` implies true, the sets $\mathcal{T}^{\texttt{p}}$ and $\mathcal{T}^{\texttt{q}}$ should be combined and aliases should be deleted. The representative axiom is shown as follows:

   $$\mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-2} \wedge \mathcal{T}^p \wedge \mathcal{T}^q \wedge \Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q \wedge \texttt{p==q} \Longrightarrow \mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-2k-2} \wedge (\mathcal{T}^p \uplus \mathcal{T}^q) \wedge \mathcal{T}^{\texttt{p->r1}} \wedge \mathcal{T}^{\texttt{q->r1}} \wedge \ldots \wedge \mathcal{T}^{\texttt{p->rk}} \wedge \mathcal{T}^{\texttt{q->rk}} \wedge \Pi \wedge \mathcal{N} \wedge \mathcal{D}$$

$\wedge Q \wedge$p->r1==q->r1$\wedge \ldots \wedge$p->rk==q->rk

in which $\uplus$ is an operation combining two sets and deleting aliases. r1,...,rk are pointer-typed fields of the structure pointed to by p. Seen from the axiom above, we need to keep on combining the sets of $\mathcal{T}^{\text{p->ri}}$ and $\mathcal{T}^{\text{q->ri}}(i = 1, \ldots, k)$ until the pointers which are equal appear in $\mathcal{D}$ or $\mathcal{N}$. Due to the limited space, the corresponding axioms are omitted.

6) Axiom schemas of boolean expressions and pointer non-membership assertions

   $\mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-1} \wedge \mathcal{T}^q \wedge \Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge$p==q$\wedge$p $\notin \mathcal{T}^q$
   $\wedge Q' \Longrightarrow$ false
   $\mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-1} \wedge \mathcal{T}^q \wedge \Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge$p!=q$\wedge$p $\notin \mathcal{T}^q$
   $\wedge Q' \Longrightarrow \mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{m-1} \wedge \mathcal{T}^q \wedge \Pi \wedge \mathcal{N} \wedge \mathcal{D}$
   $\wedge$p $\notin \mathcal{T}^q \wedge Q'$

7) Equivalence axioms of pointer non-membership assertions

   p $\notin$ (assertion1 $\wedge$ assertion2)
   $\Longleftrightarrow$ p $\notin$ assertion1 $\wedge$ p $\notin$ assertion2
   p $\notin$ ($\forall$ident:domain.assertion)
   $\Longleftrightarrow$ $\forall$ident:domain.(p $\notin$ assertion)
   p $\notin$ [lvalset] $\Longleftrightarrow$ $\forall$q:[lvalset].alias(p, q)
   p $\notin \mathcal{T}^q \Longleftrightarrow$ q $\notin \mathcal{T}^p \Longleftrightarrow$ p!=q

   Since p is a valid pointer in $\mathcal{U}$, other axioms about p$\notin${lvalset}, p$\notin${lvalset}$_{\mathcal{N}}$ or p$\notin${lvalset}$_{\mathcal{D}}$ implicates true or false can be obtained.

2.5. Extension of Specifications and Inference Rules

We use Hoare-style specifications {P}S{Q}, in which S is a syntax structure or a statement mostly, and P and Q are pre- and post- conditions respectively. Hoare-style inference rules are used to express the effects of statements to pointer information which is useful in reasoning about properties of programs.

In the original pointer logic [6], we have defined some basic operations on access path sets and predicates to figure out inference rules in detail, such as access path deletion '-', access path addition '+', prefix substitution '/', and predicate leak($\mathcal{S}$,p). Prefix substitution is used to describe that for each access path q in the given set $\mathcal{R}$, if a prefix of q is an alias of p, then q is substituted by its alias q′ where q′ is not prefixed by any alias of p; other access paths are not changed. These basic operations are still used in the inference rules below.

Since $\mathcal{U}$ is introduced, uncertainty may exist in the decision whether memory leaks. That is, when deleting access path p from a valid pointer set $\mathcal{T}$ of $\mathcal{U}$ (possibly when p is assigned), if there is a set in which all access paths are aliases of p or are prefixed by p, then there may exist memory leaks but it's not certain, because pointers in other sets of $\mathcal{U}$ may point to the same object as p does.

Here we give the extension based on the original inference rules [6]. The extended inference rules are similar to the original except sometimes it is difficult to determine whether memory leaks and to label dangling pointers.

1) Assignment statement of pointers (p=q and p=NULL)
   The original rules for Π could be almost copied as rules for $\mathcal{U}$ in most cases, except these two particular cases below because of the uncertainty in the decision whether memory leaks.

   a) An alias of p is in some valid access path set of Π or $\mathcal{U}$ and alias of q is in $\mathcal{N}$.

      $\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{U} \wedge \mathcal{N}^q \wedge \mathcal{D} \wedge Q\}$
      p=q
      $\{\mathcal{S}_1/\text{p} \wedge \ldots \wedge \mathcal{S}_{n-1}/\text{p} \wedge (\mathcal{S}^p/\text{p} - \text{p})$
      $\wedge \mathcal{U}/\text{p} \wedge (\mathcal{N}^q/\text{p} + \text{p}) \wedge \mathcal{D}/\text{p} \wedge Q/\text{p}\}$
      (if leak($\mathcal{S}^p$, p) is false based on the Ψ formed by the precondition)
      $\{\mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}_{n-1} \wedge \mathcal{T}^p \wedge \Pi \wedge \mathcal{N}^q \wedge \mathcal{D} \wedge Q\}$
      p=q
      $\{\mathcal{T}_1/\text{p} \wedge \ldots \wedge \mathcal{T}_{n-1}/\text{p} \wedge (\mathcal{T}^p/\text{p} - \text{p})$
      $\wedge \Pi \wedge (\mathcal{N}^q/\text{p} + \text{p}) \wedge \mathcal{D}/\text{p} \wedge Q/\text{p}\}$
      (test leak($\mathcal{T}^p$, p) based on the Ψ formed by the precondition)

   Note that in the former case, it requires that leak($\mathcal{S}^p$, p) is false before the assignment while in the latter case, we just test leak($\mathcal{T}^p$, p) before the assignment and give warning messages about possible memory leaks if leak($\mathcal{T}^p$, p) is true.

   b) An alias of p is in some valid access path set of Π or $\mathcal{U}$ and alias of q is in $\mathcal{D}$. The rules are similar.

2) Free rule for free(p)
   Only when an alias of p is a valid pointer, it's safe to use function free to free the object pointed to by p. When an alias of p is in some set of Π, the original rule for free(p) [6] is still usable. When an alias of p is in some set of $\mathcal{U}$, warning messages about possible dangling pointers should be given on the basis of the original free rule, because pointers in other sets of $\mathcal{U}$ may point to the same object as p does.

Other rules, including rules for assignment statement of non-pointers, malloc rule, composition/conditional/while rule, case analysis rule, frame rule and rules for function construct, are the same as those in the original pointer logic.

3. Case Study

3.1. The Schorr-Waite Algorithm

In this subsection, we take the Schorr-Waite algorithm [8] as an example to show the application of the extended pointer logic. The Schorr-Waite algorithm performs a depth-first traversal of a directed graph. It directly uses the pointers of the graph to implement backtracking. We only prove the algorithm satisfies the elementary safety policy of PointerC.

The correctness of the algorithm, *i.e.*, every node in the graph must be marked, is not proved here because the pointer logic has no distinct advantages in that case.

In the C version considered here, as in [9], we modify the boolean expressions in *while* and *if* conditions that make the boolean conditions explicit to be correctly reflected in pre- and post- conditions. Assume that short-circuit calculation is adopted in evaluation of boolean expressions to avoid excessive modification of the code. The structure of the binary directed graph nodes can be defined as:

```
typedef struct struct_node{
  boolean m, c;
  struct struct_node *l, *r;
}node;
```

where `l` and `r` are respectively pointers to the left and to the right child (they can be set to `NULL`). The field `m` is a mark of the node. The field `c` is used internally, to denote which of the children is currently being explored.

The predicate for the binary directed graph is as follows:

$$\mathsf{graph}(\texttt{p}) \triangleq \{\texttt{p}\}_{\mathcal{N}} \vee (\{\texttt{p}\}_{\mathcal{N}} \wedge \mathsf{graph}(\texttt{p->l}) \wedge \mathsf{graph}(\texttt{p->r}))$$

Pay attention to the difference between this definition and those of binary tree and binary DAG. In fact, when the graph has cycles in it, the definition will be expanded infinitely. But in our case, the fields `m` and `c` could be used to identify cycles so that the definition seems appropriate.

The proof is shown in Figure 2. Due to the limited space, the assertions at some program points are omitted.

3.2. AIO Remove Request Function in Glibc

In the previous works, we only use the pointer logic to reason about typical operations on typical data structures, such as inserting a node into a singly-linked list or a binary tree. We have not yet verified other practical programs with shared mutable data structures. Investigations in practical programs find that graphs and DAGs are mostly represented by adjacency matrix and so on. It seems that the extension of the pointer logic is trivial. However, in practical programs using linked list or binary tree, special data structures and operations are often designed so that the extended pointer logic may be helpful. In this subsection, we show how the extended pointer logic can be used in practice by illustrating a function taken from GNU C Library [13].

In GNU C Library, the asynchronous I/O operations are implemented using a request queue. The type of the request queue node is:

```
struct requestlist {
  int runnning;
  struct requestlist *last_fd;
  struct requestlist *next_fd;
  struct requestlist *next_prio;
  struct requestlist *next_run;
  //...
};
```

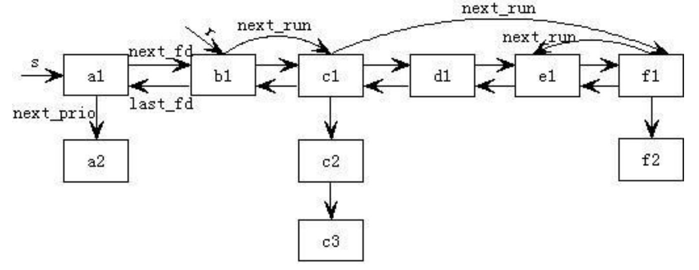

Figure 3. Request Queue and Ready Queue

Every node holds an I/O request for a file descriptor. Nodes form a doubly-linked list in order of the file descriptors via the fields `last_fd` and `next_fd`. In the list, `last_fd` of the head node and `next_fd` of the tail node are all `NULL`. If there are several I/O requests for one file descriptor, then the corresponding nodes form a singly-linked list in order of priority via the field `next_prio`. The head nodes of these singly-linked lists are in the doubly-linked list. It's obvious that equality of these pointers is certain. And then a singly-linked list via `next_run` is formed by some nodes in the request queue of which the corresponding requests are ready, so the singly-linked list is called ready queue. The equality between the pointers in ready queue and those in the doubly-linked list of request queue is uncertain.

Assume the numbers of nodes in request queue and ready queue are respectively $n+1$ and $m+1$, and `s` and `r` are respectively the head pointers of the queues (in Figure 3). The structure can be defined in the pointer logic as follows:

$\mathsf{rq}(\texttt{s,r,n,m})\texttt{=}(\forall\texttt{k:0..n-1.[s(->next_fd)}^{k}\texttt{,}$
$\texttt{s(->next_fd)}^{k+1}\texttt{->last_fd]})\wedge\texttt{[s(->next_fd)}^{n}\texttt{]}$
$\wedge\{\texttt{s->last_fd,s(->next_fd)}^{n+1}\}_{\mathcal{N}}$
`/* The doubly-linked list of request queue */`
$\wedge(\forall\texttt{k:0..n.}\mathsf{plist}\texttt{(s(->next_fd)}^{k}\texttt{->next_prio))}$
`/* Each node in the doubly-linked list links a singly-linked list */`
$\wedge\ \mathsf{rlist}\texttt{(r,m)}$ `/* Ready queue */`
$\wedge(\forall\texttt{k:0..n.}\forall\texttt{l:0..n.(k!=l}\Rightarrow\texttt{s(->next_fd)}^{k}$
$\notin\texttt{[s(->next_fd)}^{l}\texttt{,s(->next_fd)}^{l+1}\texttt{->last_fd]))}$
where

$\mathsf{plist}\texttt{(p)=}\{\texttt{p}\}_{\mathcal{N}}\vee(\{\texttt{p}\}\wedge\mathsf{plist}\texttt{(p->next_prio)}$
$\wedge\{\texttt{p->next_fd,p->last_fd,p->next_run}\}_{\mathcal{N}})$
$\mathsf{rlist}\texttt{(r,m)=}\{\texttt{r(->next_run)}^{m+1}\}_{\mathcal{N}}$
$\wedge(\forall\texttt{k:0..m.[r(->next_run)}^{k}\texttt{])}$
$\wedge(\forall\texttt{k:0..m.}\forall\texttt{l:0..m.(k!=l}\Rightarrow$
$\texttt{r(->next_run)}^{k}\notin\texttt{[r(->next_run)}^{l}\texttt{]))}$

The non-membership assertion in the definition of `rq(s,r,n,m)` expresses the pointer fields `next_fd` of nodes in the doubly-linked list are not equal to each other. Due to the limited space, some `NULL` pointers are ignored, such as the fields `next_run` of nodes which are in the doubly-linked list but not in rlist`(r,m)`, because they are not involved in

```
{graph(root)}
void schorr_waite(node * root){
  node * t, *p, *q;
  {graph(root)∧{t, p, q}_D}
  t = root; p = NULL; q = NULL; root = NULL;
  {graph(t)∧{root, p, q}_N}
  {graph(t)∧graph(p)∧{root, q}_D} /* loop invariant */
  while(p!=NULL ∨ (p==NULL ∧ t !=NULL ∧ ¬t->m)) {
    {(graph(t)∧graph(p->l)∧graph(p->r)∧[p]∧{root,q}_N)∨(graph(t->l)∧graph(t->r)∧[t]∧{p,root,q}_N∧¬t->m)}
    if(t==NULL ∨ (t!= NULL ∧ t->m)) {
      {(graph(p->l)∧graph(p->r)∧[p]∧{t,root,q}_N)∨(graph(p->l)∧graph(p->r)∧[p]∧graph(t->l)∧graph(t->r)∧[t]∧{root,q}_N)}
      if(p->c)) { /*pop*/
        q = t; t = p; p = p->r; t->r = q; q = NULL;
        {(graph(t->l)∧[t]∧graph(p)∧{root,q,t->r}_N)∨([t]∧graph(p)∧graph(t->r->l)∧graph(t->r->r)∧[t->r]∧{root,q}_N)}
      } else { /* swing */
        q = t; t = p->r; p->r = p->l; p->l = q; p->c = true; q = NULL;
        {(graph(p->r)∧graph(t)∧[p]∧{root,q,p->l}_N)
         ∨(graph(p->r)∧graph(t)∧[p]∧graph(p->l->l)∧graph(p->l->r)∧[p->l]∧{root,q}_N)}
      }
      {graph(t)∧graph(p)∧{root,q}_N} /* loop invariant */
    } else { /* push, t!=NULL∧¬t->m */
      {(graph(p->l)∧graph(p->r)∧[p]∧graph(t->l)∧graph(t->r)∧[t]∧{root,q}_N∧¬t->m)
       ∨(graph(t->l)∧graph(t->r)∧[t]∧{p,root,q}_N∧¬t->m)}
      q = p; p = t; t = t->l; p->l = q; p->m = true; p->c = false; q = NULL;
      {(graph(p->l->l)∧graph(p->l->r)∧[p->l]graph(t)∧graph(p->r)∧[p]∧{root,q}_N)
       ∨(graph(t)∧graph(p->r)∧[p]∧{root,q,p->l}_N)}
      {graph(t)∧graph(p)∧{root,q}_N} /* loop invariant */
    }
  } /* p==NULL ∧ (t==NULL ∨ (t!=NULL ∧ t->m) ) */
  {graph(t)∧{root,p,q}_N}
}
```

Figure 2. The Schorr-Waite Algorithm

the verification.

In the function

```
void __aio_remove_request
  (struct requestlist *last,
   struct requestlist *req, int all),
```

two global variables will be used:

```
/* Ready queue */
struct requestlist *runlist;
/* Request queue */
struct requestlist *requests;
```

So we have rq(requests,runlist,n,m). The numbers of nodes in the doubly-linked list of requests and in runlist are respectively $n+1$ and $m+1$.

The function has two pointer type parameters. Intuitively, the pointer last points to the predecessor of the node pointed to by req in the singly-linked list. The possible relations between them can be classified into 4 cases (in Figure 4). In order to reason in the pointer logic, the definition of rq(requests,runlist,n,m) should be expanded to include the pointers req and last. Due to the limited space, details are given in [15].

Briefly, the function removes the node N pointed to by req in all the linked relations including request queue and ready queue. Yet the function does not free the node. If all=1, all the nodes linked by req in the singly-linked list will be removed along with the node N. Otherwise, if all=0, only the node N will be removed and its successor in the singly-linked list will take its place.

Since it's unsupported in PointerC, a loop with a break statement in the original code is modified. In Figure 4, we show the code and assertions at main points of the function. More detailed proof is available at [14]. Although it has been simplified, verification by hand is still very complicated. Thus the pointer logic should be implemented in tools.

4. Related Work

We extend the pointer logic to support data structures like graphs in which equality of pointers is uncertain. More details can be found in the extended version of this paper [15]. The pointer logic essentially uses the idea of precise alias analysis in program verification. It represents pointer information of program points using access path sets and expresses the effects of statements to pointer information using Hoare-style inference rules.

Bornat also used Hoare logic to reason about properties of pointer programs [10]. He treated the heap as a pointer-indexed collection of objects, each of which is a name-indexed collection of components. An object-component

```
{rq(requests,runlist,n,m)∧(case1∨case2∨case3∨case4)}
void __aio_remove_request(struct requestlist *last,struct requestlist *req,int all){
  if(last!=NULL)
    {rq(requests,runlist,n,m)∧(case3∨case4)}
    if(all==1) last->next_prio=NULL; else last->next_prio=req->next_prio;
    {rq(requests,runlist,n,m)∧{req}}
  else{
    if(all==1∨req->next_prio==NULL){
      {rq(requests,runlist,n,m)∧(case1∨case2)}
      if(req->last_fd!=NULL) req->last_fd->next_fd=req->next_fd; else requests=req->next_fd;
      if(req->next_fd!=NULL) req->next_fd->last_fd=req->last_fd;
      {rq(requests,runlist,n-1,m)∧(∀i:0..n-1.req≠requests(->next_fd)^i)}
    } else{
      {rq(requests,runlist,n,m)∧(case1∨case2)}
      if(req->last_fd!=NULL) req->last_fd->next_fd=req->next_prio; else requests=req->next_prio;
      if(req->next_fd!=NULL) req->next_fd->last_fd=req->next_prio;
      req->next_prio->last_fd=req->last_fd; req->next_prio->next_fd=req->next_fd;
      req->next_prio->running=yes;
      {rq(requests,runlist,n,m)∧(∀i:0..n.req≠requests(->next_fd)^i)}
    }
    {(rq(requests,runlist,n-1,m)∧(∀i:0..n-1.req≠requests(->next_fd)^i))
    ∨(rq(requests,runlist,n,m)∧(∀i:0..n.req≠requests(->next_fd)^i))}
    if(req->running==yes){
      struct requestlist *runp=runlist; last=NULL;
      while(runp!=NULL∧runp!=req){last=runp; runp=runp->next_run;}
      if(runp==req) {if(last==NULL) runlist=runp->next_run; else last->next_run=runp->next_run;}
    }
    {(rq(requests,runlist,n-1,m)∧(∀i:0..n-1.req≠requests(->next_fd)^i)∧req∉rlist(runlist,m))
    ∨(rq(requests,runlist,n,m)∧(∀i:0..n.req≠requests(->next_fd)^i)∧req∉rlist(runlist,m))
    ∨(rq(requests,runlist,n-1,m-1)∧(∀i:0..n-1.req≠requests(->next_fd)^i)∧req∉rlist(runlist,m-1))
    ∨(rq(requests,runlist,n,m-1)∧(∀i:0..n.req≠requests(->next_fd)^i)∧req∉rlist(runlist,m-1))}
  }
}
{(rq(requests,runlist,n,m)∨rq(requests,runlist,n-1,m)∨rq(requests,runlist,n-1,m-1)∨rq(requests,runlist,n,m-1))
∧{req}}
case1:(last=NULL)∧(∃i:0..n.req=requests(->next_fd)^i)∧(∃j:0..m.req=runlist(->next_run)^j)
case2:(last=NULL)∧(∃i:0..n.req=requests(->next_fd)^i)∧req∉rlist(runlist,m)
case3:(∃i:0..n.last=requests(->next_fd)^i)∧(req=last->next_prio)∧req∉rlist(runlist,m)
case4:(∃i:0..n.last=requests(->next_fd)^i(->next_prio)^ip)∧(req=last->next_prio)∧req∉rlist(runlist,m)
```

Figure 4. Example of AIO Remove Request Function

reference in the heap corresponds to a double indexing, once of the heap and once of the object. He introduced axioms for object component substitution for distinct component names and used them to prove properties of programs with shared mutable data structures defined by pointers. As for inductive auxiliary definitions of data structures, Bornat defined different operators, such as in the definition of directed graph:

$$\texttt{A}*_{l,r} \triangleq \texttt{if A=nil then } \{\} \texttt{ else } \{\texttt{A}\}\cup\texttt{A.l}*_{l,r}\cup\texttt{A.r}*_{l,r} \texttt{ fi}$$

DAG is defined based on the definition of directed graph by including an exclusion set S to break cycles:

$$\texttt{A}*_{l,r,S} \triangleq \texttt{if A=nil}\vee\texttt{A}\in\texttt{S then } \{\} \texttt{ else } \{\texttt{A}\}\cup\texttt{A.l}*_{l,r,S\cup\{A\}}\cup\texttt{A.r}*_{l,r,S\cup\{A\}} \texttt{ fi}$$

In comparison, we introduce pointer non-membership assertions to break cycles. However, in Bornat's approach, it is uncertain whether nodes are sharing or separate. And Bornat assumed that data structures except directed graphs are acyclic, so that the definitions of tree and graph seem confused. Moreover, to effectively deal with the explosion of effects produced by an assignment which affects only a single location, Bornat introduced spatial-separation assertions to represent separation of objects or nodes. By contrast, the uncertainty in our extended pointer logic is restricted that pointers in the same access path set are definitely equal and objects which have complete pointer information can be unambiguously described.

Separation logic also supports data structures like directed graphs with uncertain equality of pointers. Since separation logic emphasizes that disjoint heaps have no affects on each other, it's convenient to represent the acyclic characteristic of DAGs. But without mechanisms describing the connections between disjoint heaps, it's difficult to represent equality of access paths or substructure sharing in separation logic.

To label all the sharing substructures of a specific directed graph, [11] treats the first occurrence of a subgraph encountered in a left-to-right scan as the defining instance, and cuts links to the same structure which are encountered later. These graphs are called partial graphs, *i.e.*, graphs consisting of named vertices in which not every name maps to a successor set. Partial graphs can be regarded as graphs with sharing information collected left-to-right. In [11], the heap predicate for partial DAGs in separation logic describes a left-to-right evaluation that similarly accumulates an environment. This approach overcomes the difficulties in representing sharing information in separation logic by partial graphs, but the specifications require too much information about the environment. In the case of algorithms even as simple as those which copy and dispose DAGs and graphs [11], there is much work to do.

As for safety verification, the pointer logic introduces side conditions in the typing rules so that safety verification and error detection are implemented internally. For example, there are no assertions about safety when reasoning about the Schorr-Waite algorithm in the pointer logic. By contrast, in Yang's proof of the Schorr-Waite algorithm [12], the predicate noDangling(x) is introduced to denote x is not a dangling pointer. When the CADUCEUS tool is used to prove the algorithm in [9], the predicate valid(x) is added similarly.

Moreover, the extension of Hoare logic by Bornat and separation logic are only applied to simple programs such as the Schorr-Waite algorithm and the CopyDags function. More complicated practical programs, such as the example in subsection 3.2, have not been considered. Although the proof is done by hand, we will develop plcc to support automatic verification of complicated programs.

5. Conclusion

In this paper, we extend the pointer logic designed for PointerC to support verification of pointer programs with data structures like graphs in which equality of pointers is uncertain. The inference rules of the extended pointer logic are proposed according to the actual demand so that they may not be perfect all the time.

By now, besides the need for supporting tools, another shortage of reasoning in the pointer logic is that the programs must be annotated with loop invariants, pre- and post- conditions. We will try to automatically generate loop invariants with provided declarations about which kind of data structure (such as singly-linked list and binary tree) is pointed to by the variable in the program.

Acknowledgments

This research is supported by the National Natural Science Foundation of China under Grant No.90718026 and Intel China Research Center. Any opinions, findings, and conclusions contained in this paper are those of the authors and do not reflect the views of these agencies.

References

[1] 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*, pages 55-74, July 2002.

[2] Baojian Hua, Yiyun Chen, Lin Ge, and Zhifang Wang. The PointerC programming language specification. (Technical Report) Available at: http://ssg.ustcsz.edu.cn/lss/doc/index.html.

[3] Yiyun Chen, Baojian Hua, Lin Ge, and Zhifang Wang. A Pointer Logic for Safety Verification of Pointer Programs. Chinese Journal of Computers, 31(3), March 2008.

[4] Yiyun Chen, Lin Ge, Baojian Hua, Zhaopeng Li, Cheng Liu, and Zhifang Wang. A pointer logic and certifying compiler. *Frontiers of Computer Science in China*, 1(3), pp. 297-312, Agust 2007.

[5] Yiyun Chen, Lin Ge, Baojian Hua, Zhaopeng Li, and Cheng Liu. Design of a certifying compiler supporting proof of program safety. In *Proceedings of 1st IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering*, pages 127-136, IEEE CS press, June 2007.

[6] Zhaopeng Li, Yiyun Chen, Baojian Hua, and Zhifang Wang. A Revised Pointer Logic for Verification of Pointer Programs. Accepted by *Sixth Asian Workshop on Foundations of Software (AWFS'09)*, Tokyo, Japan, April 6-8, 2009.

[7] Zhifang Wang, Yiyun Chen, Zhenming Wang, Wei Wang, and Bo Tian. An Extension to Pointer Logic for Verification. In *Proceedings of 2nd IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering*, pages 49-56, IEEE CS press, June 2008.

[8] H. Schorr and W. M. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Commun*. ACM, 10:501-506, 1967.

[9] T. Hubert and C. March. A case study of C source code verification: the Schorr-Waite algorithm. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 190-199, IEEE CS press, 2005.

[10] R. Bornat. Proving pointer programs in Hoare logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, pages 102-126, July 03-05, 2000.

[11] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *Proceedings of the Conference on Semantics, Program Analysis, and Computing Environments (SPACE'04)*. Venice, Italy. 2004.

[12] H. Yang. An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In F. Henglein, J. Hughes, H. Makholm, and H. Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, pages 41-68. IT University of Copenhagen, 2001.

[13] The GNU C Library. http://www.gnu.org/software/libc/.

[14] An example of reasoning in extended pointer logic: AIO Remove Request Function in Glibc. Available at: http://ssg.ustcsz.edu.cn/lss/papers/.

[15] Hongjin Liang, Yu Zhang, Yiyun Chen, Zhaopeng Li, Baojian Hua. A Pointer Logic Dealing with Uncertain Equality of Pointers (Extended Version). Available at: http://ssg.ustcsz.edu.cn/lss/papers/.