

# Asserting Memory Shape using Linear Logic

Frances Spalding      Limin Jia

Department of Computer Science, Princeton University  
35 Olden Street, Princeton, NJ 08544  
{frances, ljia}@cs.princeton.edu

## ABSTRACT

Contracts and assertions are accepted as an important method for improving software reliability. However, existing systems do not provide clean ways to describe conditions based on memory shape. We present a method for elegantly specifying memory shape invariants using specifications in linear logic and then dynamically verifying these specifications using the linear logic programming language LolliMon.

## 1. INTRODUCTION

The recent research on separation logic by O’Hearn, Reynolds, Yang, et al. [18, 11, 19] has made significant progress in the static verification of the correctness of pointer programs. One of the basic ideas of separation logic is to use the multiplicative connective  $*$  to describe the disjointness of two separate pieces of memory. Separation logic can describe aliasing and shape invariants of the program store elegantly when compared with conventional logic. For example, if we wish to use a conventional logic to state that the heap can be divided into two pieces and one piece can be described by  $F_1$  and one by  $F_2$ , then we would need to say  $F_1 \wedge F_2 \wedge (S_1 \cap S_2 = \emptyset)$  where  $S_1$  and  $S_2$  are the sets of program locations that  $F_1$  and  $F_2$  respectively depend upon. As the number of disjoint memory chunks increases, the separation logic formula remains relatively simple:  $F_1 * F_2 * F_3 * F_4$  represents four separate pieces of the store. On the other hand, the related classical formula becomes increasingly complex:

$$\begin{aligned} & F_1 \wedge F_2 \wedge F_3 \wedge F_4 \\ & \wedge (S_1 \cap S_2 = \emptyset) \wedge (S_1 \cap S_3 = \emptyset) \wedge (S_1 \cap S_4 = \emptyset) \\ & \wedge (S_2 \cap S_3 = \emptyset) \wedge (S_2 \cap S_4 = \emptyset) \wedge (S_3 \cap S_4 = \emptyset) \end{aligned}$$

The end result is that while in theory it is possible to reason about memory in conventional classical logic, in practice invariants concerning unaliased data structures can quickly grow to an unmanageable size.

Separation logic has already been used to prove the cor-

rectness of programs that manipulate complex recursive data structures. One of the most impressive results is Birkedal et al.’s proof of the correctness of a copying garbage collector [5].

However, like most static verification methods, performing verification using separation logic requires the entire program. Such a verification process is rather heavy-weight and it does not allow statically verified code to be linked with untrusted code because the statically checked invariants are not guaranteed to hold at the trusted/untrusted boundary. Furthermore, automated proving tools for first-order separation logic are nonexistent, although developing such tools is an active area of research. Without the help of a theorem prover or logic programming engine to discharge proof obligations, verification using first order separation logic has to be done by hand, which is too heavy-weight for ordinary programmers to incorporate into their everyday programming or debugging.

On the other hand, people have studied dynamic contracts and assertions since the late 1960s [6, 9]. Contracts and assertions are language constructs that provide runtime checks to ensure that a specified property of the program holds, and thus can help greatly in the debugging process. Many current languages include these features, including Eiffel [16] and Java [1].

In this paper we look into exploiting the powerful connectives of substructural logic to develop a dynamic contract system for memory. We allow programmers to insert dynamic assertions about the invariants of complex recursive data structures. For instance, a programmer writing a linked list library can insert an assertion at the end of the delete function to check that the resulting list does not violate the invariants of a linked list. Assertions such as this can be tremendously useful in debugging programs that manipulate complex data structures. Instead of defining such checking functions in native code, we propose to define the invariants in a declarative linear logic programming language, and then utilize a linear logic programming engine such as Lolli [10] or LolliMon [14] to check the assertions.

The advantage of using a logic programming language as an assertion language is that the assertions are expressed at a high-level of abstraction and have a formal semantics. The assertions in our system are at a much higher level than a “roll your own” function that a programmer might write in native code to check invariants about memory shape. In addition, such a function normally has to keep track of the set of locations that have been traversed in order to check disjointness properties of data structures. By using linear logic,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

the logic engine infrastructure takes care of this relatively complex task automatically.

In our previous work [2, 12] we have already shown how to use intuitionistic linear logic to describe the program store. We chose linear logic instead of separation logic because the multiplicative connectives in linear logic have similar properties as the multiplicative connectives in separation logic, and there are existing implementations of linear logic programming languages [10, 14] that we can use.

Finally, dynamic and static checking are complementary to each other. We hope to be able to integrate our dynamic verification system with static analysis.

There are two main contributions of our work.

- We have built an interpreter for MiniC, a language that is essentially a subset of C extended with a logical assertion language. The MiniC interpreter interfaces with a linear logic programming engine to dynamically check assertions about the shape invariants of the program heap.
- We have shown how to dynamically verify the shape of many commonly used recursive data structures including single linked lists, circular lists, doubly linked lists, trees, and other more complicated data structures such as red-black trees and b-trees.

The rest of the paper is organized as follows. In Section 2 we first briefly review the syntax and the store semantics of linear logic, then give an overview of our MiniC system by explaining two examples. In Section 3 we introduce an indexed memory model for our logic and prove the soundness of assertions. In Section 4 we provide implementation details of the MiniC Language and how it interfaces with LolliMon. In Section 5 we discuss how to represent data structures that include aliasing. In Section 6 we explore example code for red-black trees. And finally, in Section 7 we discuss related and future work.

## 2. OVERVIEW

In this section we first review the basics of linear logic. Then we will go over an example program about structs to illustrate the components of a program in our system. Next we briefly introduce LolliMon and we finish the struct example. Lastly, we show how to verify singly linked lists in our system.

### 2.1 Linear Logic

We use intuitionistic linear logic to describe the program store. The syntactic constructs of linear logic are listed below. We use  $\ell$  to denote memory locations, and  $v$  to denote values. Memory locations and values are both infinite subsets of integers. We treat each memory word as one unit, so we write  $(\ell+1)$  for the address immediately after  $\ell$ . The basic predicates include  $(\text{lin } \ell v)$ , arithmetic predicates  $P_A$ , and user defined predicates  $P$ . The connectives include both the multiplicative and additive linear connectives as well as the unrestricted modality  $!$ .

$$\begin{aligned} \text{Formulas } F ::= & P \mid P_A \mid (\text{lin } \ell v) \mid \mathbf{1} \mid F_1 \otimes F_2 \\ & \mid F_1 \multimap F_2 \mid \top \mid F_1 \& F_2 \mid \mathbf{0} \mid F_1 \oplus F_2 \\ & \mid !F \mid \exists x.F \mid \forall x.F \end{aligned}$$

Predicate  $(\text{lin } \ell v)$  describes a memory that contains only one location  $\ell$  with the contents  $v$ . The multiplicative connective  $\otimes$  (similar to  $*$  in separation logic) separates the

memory into two disjoint pieces. For example, there is no memory that satisfies formula  $(\text{lin } \ell 3) \otimes (\text{lin } \ell 3)$ , because that would require the same location  $\ell$  to be in two disjoint pieces of memory. The connective  $\mathbf{1}$  is the unit of the multiplicative conjunction and it describes empty memories. Formula  $F_1 \multimap F_2$  describes memories whose union with memories described by  $F_1$  satisfy  $F_2$ . Formula  $F_1 \& F_2$  describes memories that satisfy both  $F_1$  and  $F_2$ . The connective  $\top$  is the unit of the additive conjunction and it describes all memories. Formula  $F_1 \oplus F_2$  describes memories that satisfy either  $F_1$  or  $F_2$ . Connective  $\mathbf{0}$  is the falsehood and no memory satisfies it. The semantics of the unrestricted modality  $!$  force  $F$  to be valid with empty memory. Arithmetic predicates  $P_A$  include equality and less than relationship over integer expressions. For example,  $(x = 2) \oplus (2 < x)$  is true if  $x$  is greater than or equal to two.

### 2.2 Example: Structs

The sample program `pair.minic` in Figure 1 is written in our prototype system, MiniC, whose syntax is a subset of ANSI-C extended with syntax for defining clauses and asserting formulas in LolliMon. The LolliMon declarations in double brackets at the very beginning of the program are user-defined clauses for predicates describing memory shapes of concern. We delay explaining them until after the next sub-section, and focus on the C program for now. In the `main` function, the programmer allocates a struct of type `pair_tp`, then calls the `copy` function, which is defined in a different source file written by another programmer. From the comments, we know that `copy` is supposed to perform deep copying. However, the programmer writing the `main` function did not write the `copy` function, so he wants to check that upon return, the specifications in the comments are met.

The last statement in the `main` function is the `assert` statement. The linear logic formula in the double brackets of the `assert` statement describes the desired shape of the memory.

### 2.3 LolliMon

We use LolliMon [14], a monadic concurrent linear logic programming language, as our assertion language. LolliMon extends the linear logic programming language Lolli [10] with synchronous monads. We have not used the monads in our examples, so we will omit them here.

In LolliMon, we can define clauses consisting of a head and a body just as we would in Prolog. Goal formulas can be queried over the set of defined clauses. The syntax for writing clauses and queries in LolliMon differs from the formal linear logic syntax in that we write “,” in place of “ $\otimes$ ” and “;” in place of “ $\oplus$ ”. Other operators are immediately recognizable in their textual form. Following the style of Prolog, we write clauses as inverted implication: `Head o- Body`. If the body of the clause is unrestricted, we use the unrestricted implication `<=` in place of the linear implication `o-1`. LolliMon also provides the built-in predicate `is`, which evaluates its integer arguments and then checks that they are equal, the built-in list operator `::`, and the term `nil` for the empty list. Below is a sample program in LolliMon.

<sup>1</sup>Formula  $F_1 \Rightarrow F_2$  is equivalent to  $!F_1 \multimap F_2$ .

```

[[
struct L nil.
struct L (V::Y) o-
    Lin L V, L1 is L + 1, struct L1 Y.
]]

struct pair_tp { int x; int y;};

/* copy is a deep copy function that takes
 * a pointer to a pair_tp struct, copies
 * the contents into a newly allocated
 * struct and returns the pointer of the
 * new struct */
extern struct pair_tp *copy(struct pair_tp *x);

int main(){
    struct pair_tp *pair2;
    struct pair_tp *pair1
        = malloc(sizeof(struct pair_tp));
    pair1->x = 100;
    pair1->y = 200;
    pair2 = copy(pair1);

/* pair2 and pair1 should refer to
 * different locations with the same data */
    assert([[struct $pair1 (X::Y::nil),
            struct $pair2 (X::Y::nil)]]);
}

```

Figure 1: pair.minic

```

1 struct L nil.
2 struct L (V::Y) o-
3     Lin L V, L1 is L + 1, struct L1 Y.

5 #linear lin 10 100, lin 11 200.
6 #linear lin 20 100, lin 21 200.

8 #query 1 1 1 (struct 10 (X::Y::nil),
9     struct 20 (X::Y::nil)).

```

We defined two clauses for predicate `struct` in lines 1 through 3. Predicate `struct L X` means that `X` is the list of values stored in the memory chunk starting from address `L`. The first clause handles the base case where there are no more elements in the list. In the second clause, the memory starting from address `L` points to the list of elements `V :: Y` if `L` points to the first element (`lin L V`) and the next location `L+1` is a struct that has the rest of the elements (`struct L1 Y`)<sup>2</sup>. On line 5, we define a linear clause that states that location 10 contains integer 100 and location 11 contains integer 200. Line 6 similarly declares that location 20 contains 100 and location 21 contains 200. The keyword `#linear` enforces that the clause must be used exactly once in proving the goal. Lines 8 and 9 contain the query to be solved. The first three parameters indicate the number of expected solutions, the maximum number of solutions, and the number of attempts that should be made to try to prove this

<sup>2</sup>Notice that locations are treated at a high level of abstraction. Adjacent fields in a struct or array are offset by 1.

query. The last argument is the formula we are attempting to prove.

In this case, the queried formula asks if there exist two disjoint pieces of memory and some data `X` and `Y` such that the first piece of memory starts from location 10 and contains two elements `X` and `Y`, and the second piece of memory starts from location 20 and contains the same pair of values `X` and `Y`. This query succeeds because it uses each of the linear resources exactly once and the logical variable `X` is unified with 100 and `Y` with 200.

## 2.4 Example: Structs, Continued

As we saw in the sample MiniC code `pair.minic`, programmers define clauses that specify the shape and other invariants of their data structures at the very beginning of the program. They can then insert assertions based on these definitions at any point in the code. Intuitively, at run time when an assert is seen, the definitions of the clauses together with the formula describing the program memory are given as the logical context to the LolliMon engine, and the formula to be asserted is sent to the engine as the query formula to be executed against the context. If the query is proved by the logic engine, the program will continue running; if it fails, the program will be aborted.

Now we continue the example of program `pair.minic` in Figure 1. Assume the `copy` function has the implementation below:

```

/* copy is a deep copy function that takes
 * a pointer to a pair_tp struct, copies
 * the contents into a newly allocated
 * struct and returns the pointer of the
 * new struct */
struct pair_tp *copy(struct pair_tp *p){
    struct pair_tp *dup
        = malloc(sizeof(struct pair_tp));
    dup->x = p->x;
    dup->y = p->y;
    return dup;
};

```

When the assert in `main` is reached, the heap of this program contains two structs `pair1` and `pair2`. `pair1` is allocated by the `main` function, and `pair2` is allocated by the `copy` function. Assuming `pair1` is allocated at location `L1` and `pair2` is allocated at location `L2`, then the formula describing the heap is

$$\begin{aligned}
 &(\text{lin } L1 \ 100), (\text{lin } L1 + 1 \ 200), \\
 &(\text{lin } L2 \ 100), (\text{lin } L2 + 1 \ 200)
 \end{aligned}$$

After the variable names are replaced by the values they are bound to, the assert formula becomes

$$\text{struct } L1 \ (X :: Y :: \text{nil}), \text{ struct } L2 \ (X :: Y :: \text{nil})$$

The above formula is checked against the clauses defined in the first three lines of `pair.minic` and the formula describing the heap. Assuming that `L1` is 10 and `L2` is 20, then the logic program invoked to check the assertion in this MiniC program is exactly the program in the previous subsection, so the assertion passes.

Suppose on the other hand that the `copy` function is erroneously implemented as follows:

```

/* copy is a deep copy function that takes
 * a pointer to a pair_tp struct, copies
 * the contents into a newly allocated
 * struct and returns the pointer of the
 * new struct */
struct pair_tp *copy(struct pair_tp *p){
    return p;
};

```

When the assert is reached, the heap of this program has only one struct `pair1` and `pair2` is an alias of `pair1`. The assertion is expanded to

```
struct L1 (X :: Y :: nil), struct L1 (X :: Y :: nil)
```

And the formula describing the current heap is

```
(lin L1 100), (lin L1+1 200).
```

In this case the assertion will fail because there are not enough linear resources to prove that there are two disjoint structs.

## 2.5 Example: Linked Lists

Here we show how to define predicates to describe the invariants of non-circular singly linked lists. Predicate `(l1ist L)` means the memory chunk starting from location `L` is a singly linked list. A location `L` is a list if either it is null (`0`) or it contains data `Head` and the value, `Tail`, in the next location is also a list.

```
l1ist L o- (L is 0);
          (struct L (Head :: Tail :: nil),
           l1ist Tail).
```

In the MiniC program `l1ist.minic` in Figure 2, the LolliMon definitions are between line 1 and 9. In addition to the above clause definition, the LolliMon definitions list program also includes the type declaration of the `l1ist` predicate (line 2) and the mode declaration (line 4), which says whether each parameter is an input or an output. Following the LolliMon definitions are the MiniC definitions to declare a list node structure, `struct node_tp` (line 11–15).

The memory in Figure 3 contains a linked list. The `main` function first constructs a list matching the one in Figure 3 (line 20–29). Assume that `list3` is allocated at location  $\ell_3$ , `list2` at  $\ell_2$ , and `list1` at  $\ell_1$ . The programmer then asserts at line 32 that `list1` is a linked list (`[[l1ist $list1]]`). The assertion formula becomes `(l1ist  $\ell_1$ )` after replacing the variable with its value. Since  $\ell_1$  is not 0, the logic engine continues to solve the subgoal on line 7 and 8. The linear resources corresponding to memory  $m_1$  are used to determine that `Head` is 1 and `Tail` is  $\ell_2$ . The logic engine then attempts to prove that  $\ell_2$  is a list using the remainder of the memory  $m_2 \uplus m_3$ . This in turn is reduced to proving that  $\ell_3$  is a list using  $m_3$ . The base case is reached when proving that 0 is a list using no memory. Since all subgoals are solved, the assertion passes.

On line 35, the `main` function changes the last element in the list to point to the first, resulting in a circular list. The second assert, at line 38, begins much like the first. However, when the subgoal of proving  $\ell_3$  is a list is reached, the logic engine tries to use the resources corresponding to  $m_3$  to prove that  $\ell_3$  is a list. The next step would be to prove that  $\ell_1$  is a list using no linear resource. Unlike the first assert,

```

/* an l1ist is a non-circular linked list */
1  [[
2  l1ist: int -> o.

4  #mode l1ist +L.

6  l1ist L :- (L is 0);
7          (struct L (Head::Tail::nil),
8          l1ist Tail).
9  ]]

11 struct node_tp {
12     int data;
13     struct node_tp* next;
14 };
15 typedef struct node_tp* list_tp;

17 int main() {
18     list_tp list1, list2, list3;

20     /* build a list of length 3 */
21     list3 = malloc(sizeof(struct node_tp));
22     list3->data = 3;
23     list3->next = 0;
24     list2 = malloc(sizeof(struct node_tp));
25     list2->data = 2;
26     list2->next = list3;
27     list1 = malloc(sizeof(struct node_tp));
28     list1->data = 1;
29     list1->next = list2;

31     /* check the list is well-formed */
32     assert([[l1ist $list1]]);

34     /* make the list circular */
35     list3->next = list1;

37     /* this assert fails */
38     assert([[l1ist $list1]]);

40     return 0;
41 }

```

Figure 2: A simple MiniC program `l1ist.minic`.

where we can prove that 0 is a list without using any linear resources, proving that  $\ell_1$  is a list requires us to consume the memory  $m_1$ . However, this resource has already been used at the very beginning of the proof, so this assertion is unable to succeed and the MiniC interpreter throws an `Assert Failed` exception.

The actual output from running `l1ist.minic` is shown in Figure 4. Each time an assertion is encountered, the line number and formula to be queried are printed. The next line contains the actual query with the program variables substituted with the actual values. If the assertion fails, the stack and heap are printed to aid in debugging. The value of `m(2u)` seen in the example heap is the tag created by the `malloc` function specifying that the next two locations are allocated (in use).

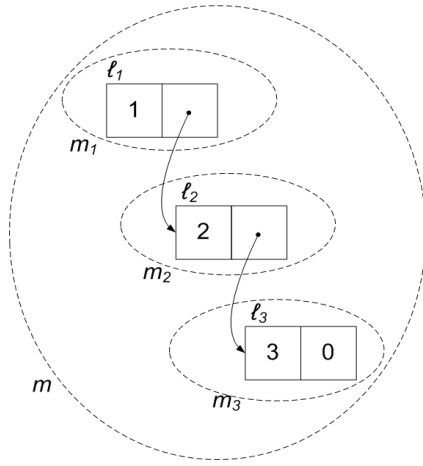


Figure 3: Example memory containing a linked list.

```

% ./runMiniC tests/llist.minic

Checking assertion 32.4 - 32.28:
  [[llist $list1]]
Looking for 1 solutions to query: llist 1025
Attempt 1, Solution 1 with []
Success.
Time consumed    0.02

Checking assertion 38.4 - 38.28:
  [[llist $list1]]
Looking for 1 solutions to query: llist 1025
Failed to find 1 solutions within 1 attempts.
Time consumed    0.03    seconds.

Stack:
0x0000: 0x0407
0x0001: 0x0404
0x0002: 0x0401

Heap:
0x0400: m(2u)  0x0401: 3    0x0402: 0x0407
0x0403: m(2u)  0x0404: 2    0x0405: 0x0401
0x0406: m(2u)  0x0407: 1    0x0408: 0x0404

Assertion [[llist $list1]] Failed
      at Position 38.4 - 38.28

```

Figure 4: The output from running `llist.minic`.

### 3. MEMORY SEMANTICS OF LINEAR LOGIC

In this section, we first explain the formal syntax of recursively defined predicates, and then we introduce an indexed semantics of the intuitionistic linear logic with recursive definitions. Next we present our formal result, the theorems of the soundness of assertions. Since LolliMon is a fragment of intuitionistic linear logic, all the soundness results of this section carry over to LolliMon.

#### 3.1 Recursive Definitions

In the list example in the previous section, the body of

the clause defining predicate `llist` contains the predicate `llist` itself. We use  $I$  to denote the definition of a recursive predicate, and  $Is$  to denote the list of such clauses.

$$\begin{aligned}
 \text{Pred def } I &::= \forall x_1 \dots \forall x_m. (F \multimap P x_1 \dots x_n) \\
 \text{Pred defs } Is &::= \cdot \mid I, Is
 \end{aligned}$$

Each definition  $I$  corresponds to a clause definition in LolliMon with the free logical variables universally quantified. Predicate  $(P x_1 \dots x_n)$  corresponds to the head of a clause and  $F$  corresponds to the body of the clause. We also call the body formula  $F$  an unrolling of the head predicate  $(P x_1 \dots x_n)$ . For example, below is the clause definition for `llist` in Section 2.5 given in the form of  $I$ :

$$\begin{aligned}
 \forall l. \forall head. \forall tail. \quad & ( (l = 0) \\
 & \oplus (\text{struct } l \text{ head} :: tail :: nil \otimes (\text{llist } tail)) ) \\
 & \multimap \text{llist } l
 \end{aligned}$$

#### 3.2 An Indexed Memory Model for Linear Logic

In order to relate assertion formulas to memory shape, we define an indexed memory semantics for linear logic. The memory semantics of the multiplicative and additive connectives of linear logic is very similar to those of the separation logic [19]. The indexing scheme is inspired by the indexed semantics model for recursive types developed by Appel et al. [4] and the indexed memory model in recent work by Morrisett et al. [17]. A memory  $m$  maps locations  $\ell$  to values  $v$ . We write  $dom(m)$  to denote the set of locations in the domain of memory  $m$  and  $m(\ell)$  to denote the value stored in location  $\ell$ . We use  $m_1 \# m_2$  to denote that two memories  $m_1$  and  $m_2$  have disjoint domains. Lastly  $m_1 \uplus m_2$  is the union of  $m_1$  and  $m_2$  if  $m_1 \# m_2$ , otherwise it is undefined.

The indexed semantic judgments are inductively defined over the index  $n$  and the structure of the formula. Judgment  $m \models_{Is}^n F$  states that given the set of predicate definitions  $Is$ , memory  $m$  can be described by formula  $F$  with  $n$  steps of approximation. The semantics of formulas is given in Figure 5.

The semantics for most of the linear logic connectives and predicate  $(\text{lin } \ell v)$  and  $P_A$  are straightforward and they are not affected by the indexing scheme. In those cases, the index number  $n$  is just carried around in the semantic judgments.

One interesting case is the semantics of a recursively defined predicate such as the `struct` and `llist` predicates we have shown in the previous examples. Intuitively, the index  $n$  can be seen as the number of unrolling steps of the recursively defined predicates. When the index is 0, meaning we do not unroll the predicate at all and cannot examine the definition, all memories satisfy the predicate. A memory  $m$  satisfies the predicate  $(P t_1 \dots t_n)$  at the  $n^{\text{th}}$  unrolling, when  $m$  satisfies the clause body at the  $(n-1)^{\text{th}}$  unrolling. A predicate unrolls to  $F[t_1/x_1] \dots [t_n/x_n]$  when  $(\forall x_1 \dots \forall x_m. (F \multimap P x_1 \dots x_n)) \in Is$ .

Now we use the semantics of a list predicate  $(\text{llist}' \ell)$  to illustrate the idea of indexing. This definition is a simplified version of the definition of `llist` where the definition of `struct` is expanded and the data field is dropped. The definition of `llist'` is given below:

$$\forall l. \forall x. ( (l = 0) \oplus (\text{lin } l \ x \otimes (\text{llist}' x)) ) \multimap \text{llist}' l$$

We use  $S_n$  to represent the set of memories that satis-

$m \vDash_{Is}^n F$

- $n = 0$ ,  $m \vDash_{Is}^n F$  for all memory  $m$ .
- $n \geq 1$ ,
  - $m \vDash_{Is}^n (\text{lin } \ell v)$  iff  $\text{dom}(m) = \ell$  and  $m(\ell) = v$
  - $\cdot \vDash_{Is} P_A$  iff  $P_A$  is true
  - $m \vDash_{Is}^n P \ t_1 \dots t_n$  iff  $(\forall x_1 \dots \forall x_n. (F \multimap P \ x_1 \dots x_n)) \in Is$ , and  $m \vDash_{Is}^{n-1} F[t_1/x_1] \dots [t_n/x_n]$
  - $m \vDash_{Is}^n \top$
  - $m \vDash_{Is}^n \mathbf{1}$  iff  $\text{dom}(m) = \emptyset$
  - $\nexists m. m \vDash_{Is}^n \mathbf{0}$
  - $m \vDash_{Is}^n F_1 \ \& \ F_2$  iff  $m \vDash_{Is}^n F_1$  and  $m \vDash_{Is}^n F_2$
  - $m \vDash_{Is}^n F_1 \ \multimap \ F_2$  iff for all  $m'$  and  $m \# m'$ , and for all  $j$ ,  $0 \leq j \leq n$  such that  $m' \vDash_{Is}^j F_1$  implies  $m \uplus m' \vDash_{Is}^j F_2$
  - $m \vDash_{Is}^n F_1 \ \otimes \ F_2$  iff there exists  $m_1$  and  $m_2$  such that  $m = m_1 \uplus m_2$  and  $m_1 \vDash_{Is}^n F_1$  and  $m_2 \vDash_{Is}^n F_2$
  - $m \vDash_{Is}^n !F$  iff  $\text{dom}(m) = \emptyset$  and  $\cdot \vDash_{Is} F$
  - $m \vDash_{Is}^n F_1 \ \oplus \ F_2$  iff either  $m \vDash_{Is}^n F_1$  or  $m \vDash_{Is}^n F_2$
  - $m \vDash_{Is}^n \forall x : \mathbf{K}. F'$  iff for all  $a \in \mathbf{K}$   $m \vDash_{Is}^n F'[a/x]$
  - $m \vDash_{Is}^n \exists x : \mathbf{K}. F'$  iff there exists  $a \in \mathbf{K}$  such that  $m \vDash_{Is}^n F'[a/x]$

**Figure 5: Semantics of Formulas**

fies  $(\text{llist}' \ell)$  at the  $n^{\text{th}}$  approximation ( $S_n = \{m \mid m \vDash_{Is}^n \text{llist}' \ell\}$ ).

- $S_0 =$  The set of all memory
- $S_1 =$  The set of all memory
- $S_2 = \{m \mid m = \emptyset \text{ or } m = (\ell \mapsto v) \uplus m_0 \text{ where } m_0 \in S_1\}$
- $S_3 = \{m \mid m = \emptyset \text{ or } m = (\ell \mapsto 0) \text{ or } m = (\ell \mapsto \ell_1) \uplus (\ell_1 \mapsto v) \uplus m_0 \text{ where } m_0 \in S_1\}$
- $S_4 = \{m \mid m = \emptyset \text{ or } m = (\ell \mapsto 0) \text{ or } m = (\ell \mapsto \ell_1) \uplus (\ell_1 \mapsto 0) \text{ or } m = (\ell \mapsto \ell_1) \uplus (\ell_1 \mapsto \ell_2) \uplus (\ell_2 \mapsto v) \uplus m_0 \text{ where } m_0 \in S_1\}$

When the index  $n$  is 0, we have the least precise idea of what the memories that satisfy  $\text{llist}' \ell$  look like, so set  $S_0$  is the set of all memories. At one step of approximation, set  $S_1$  contains memories that satisfy the unrolling of  $(\text{llist}' \ell)$  at  $0^{\text{th}}$  approximation, so  $S_1$  is also the set of all memories. We can see that set  $S_2$  contains the exact memories that satisfy lists of length 0; set  $S_3$  contains the exact memories that satisfy lists of length 0 and 1; set  $S_4$  contains the exact memories that satisfy lists of length 0, 1, and 2; so on and so forth. As the index grows bigger, the set of memories that satisfy the formula becomes smaller, and the semantics judgment becomes more precise. As the index  $n$  approaches positive infinity, we reach the greatest fixed point.

Another case worth discussing is the semantics judgment

of linear implication  $m \vDash_{Is}^n F_1 \multimap F_2$ . Because  $F_1$  is on the negative position, we have to define the semantics so that for all approximation steps up to  $n$ , the union of  $m$  and any memory  $m'$ , that satisfies  $F_1$ , satisfies  $F_2$ .

The following lemma states that the  $n^{\text{th}}$  approximation is always more precise than any  $j$  steps of approximation where  $j$  is strictly less than  $n$ . This lemma is crucial in the soundness proofs in Section 3.4.

### Lemma 1 (Downward Closure)

For all  $n \geq 1$ , if  $m \vDash_{Is}^n F$  then for all  $j$ ,  $0 \leq j < n$ ,  $m \vDash_{Is}^j F$ .

PROOF. By induction on the index  $n$  and the structure of  $G$ .  $\square$

### 3.3 Soundness of Logical Deduction

The sequent calculus of linear logic is of the form:  $\Gamma; \Delta \longrightarrow F$ . Context  $\Gamma$  contains unrestricted resources, and context  $\Delta$  contains linear resources. The unrestricted resources can be used any number of times to prove  $F$ , and each of the linear resources must be used exactly once. The sequent calculus rules are provided for reference in Appendix A.

The actual logical deduction rules for LolliMon are more complicated than those of linear logic due to the addition of monads. However, LolliMon is sound with regard to the sequent calculus rules in Appendix A. Therefore, in order to show the soundness of the LolliMon logical deductions with regard to our memory model, we only need to prove the soundness of the sequent calculus rules for intuitionistic linear logic with regard to the model.

First, we define the semantics of logical contexts. We write  $!\Gamma$  to represent the resulting formula after wrapping each formula in  $\Gamma$  in the unrestricted modality  $!$  and then tensoring these together. The notation  $\otimes \Delta$  stands for the formula resulting from tensoring together each formula in  $\Delta$ .

$$\begin{aligned} !(\cdot) &= \mathbf{1} & !(\Gamma, F) &= !F \otimes !\Gamma \\ \otimes(\cdot) &= \mathbf{1} & \otimes(\Delta, F) &= F \otimes (\otimes \Delta) \end{aligned}$$

A memory  $m$  is described by the unrestricted context  $\Gamma$  and the linear context  $\Delta$  if  $m$  is described by the formula resulting from wrapping each formula in  $\Gamma$  with  $!$  and then tensoring these together along with the contents of  $\Delta$ :

$$m \vDash_{Is}^n \Gamma; \Delta \stackrel{\text{def}}{=} m \vDash_{Is}^n !\Gamma \otimes \otimes \Delta$$

We proved that if memory  $m$  is described by contexts  $\Gamma$  and  $\Delta$ , then  $m$  is also described by the logical consequence of  $\Gamma$  and  $\Delta$ .

### Theorem 2 (Soundness of Logical Deduction)

If  $\Gamma; \Delta \longrightarrow F$  and for all  $n \geq 0$ ,  $m \vDash_{Is}^n \Gamma; \Delta$  implies  $m \vDash_{Is}^n F$ .

PROOF. By induction on the structure of  $\Gamma; \Delta \longrightarrow F$ .  $\square$

### 3.4 Soundness of Assertions

Our main technical result is a proof that if an assertion of formula  $F$  succeeds, then the current memory state can be described by  $F$ .

When an assertion is reached, the user-defined inductive definitions  $Is$  are dumped into the unrestricted context  $\Gamma$ . The formulas describing each allocated location in the current program heap are dumped into the linear context  $\Delta$ .

We use the notation  $\text{Locs}(m)$  to represent the set of formulas created by encoding each live heap location  $\ell$  that contains value  $v$  into its describing formula  $(\text{lin } \ell v)$ .

We first show (Lemma 3) that the recursive definitions  $I_s$  are valid with an empty memory.

**Lemma 3**

For all  $n \geq 1$ ,  $\cdot \models_{I_s}^n I_s$

PROOF. By the semantics of  $P t_1 \dots t_n, \neg, \forall$ , and Lemma 1.  $\square$

Next we show the correctness of the encoding of memory  $m$  by  $\text{Locs}(m)$ . In other words, memory  $m$  can be described by the tensoring of all predicates in  $\text{Locs}(m)$ .

**Lemma 4**

For all  $n \geq 0$ ,  $m \models_{I_s}^n \otimes(\text{Locs}(m))$

PROOF. By the semantics of  $(\text{lin } \ell v)$  and  $\otimes$ .  $\square$

Finally, we show that if an assertion succeeds, then the current memory  $m$  can be described by the asserted formula. We have proven that the current memory can be described by of the unrestricted context built using the recursive predicates and the restricted context built by dumping the current memory locations. Therefore, we can invoke the soundness of logical deduction theorem (Theorem 2) and conclude that the current memory can be described by the asserted formula.

**Theorem 5 (Soundness of Assertions)**

For all  $n \geq 1$ , if  $I_s; \text{Locs}(m) \longrightarrow F$  then  $m \models_{I_s}^n F$ .

PROOF. By Lemma 3, Lemma 4, and Theorem 2.  $\square$

**4. IMPLEMENTATION**

The MiniC system consists of a simple lexer, parser, and interpreter for a subset of C and an interface to the implementation of the logic programming language LolliMon. When the interpreter encounters an assertion, LolliMon is called to verify the assertion.

**4.1 The MiniC Language**

The MiniC language is a subset of C including basic control flow constructs, pointers, structs, unions, and enums with the addition of inductive definitions and assert statements in LolliMon.

A MiniC program begins with a set of clause definitions in LolliMon. These definitions are enclosed in double square brackets. The implementation automatically includes some basic predicates such as `lin` and `struct`. (As we saw in Section 2.2, the `struct L V` predicate describes a flat layout of the list of values  $V$  in the memory starting from address  $L$ .)

Next there is a sequence of top level declarations that can include global variables, struct and union definitions, type definitions, function declarations, and enumerations. The final piece of every MiniC program is a main function.

An `assert` statement takes the formula to be asserted in double square brackets. Program variables may be included in the formula by prefacing them with a dollar sign.

The MiniC interpreter is written in OCaml. It is completely standard, except for the interpretation of `assert` statements. When an `assert` is reached, it calls the logic engine

LolliMon with three pieces of information: the user-defined definitions from the top of the program, the current state of the heap, and the formula that needs to be checked. If LolliMon succeeds in proving the formula from the provided resources, the interpreter simply continues. If not, the interpreter halts with an `Assertion Failed` exception.

**4.2 The Logic Engine**

We use LolliMon [14] as the logic programming language to check assertions in MiniC programs. The backward-chaining operational semantics of LolliMon give a natural interpretation of the logical connectives as goal-directed search instructions<sup>3</sup>.

Because linear logic requires that the formulas in the linear context have to be used exactly one, the resource management for a linear logic programming language can be quite complicated. The resource management of LolliMon implements the Tag Frame Fast System [15]. Each formula in the context is assigned a special tag to indicate the usage of this formula. The linear logical context is globally available throughout the proof, and only the tags of the formulas are marked when they are used in the proof. The Tag Frame Fast System manage to make most context manipulating operations take constant time, except the `pick` rule. The `pick` rule requires going through the context linearly to choose a formula to use in order to prove the goal predicate. Next we will explain how we modified the implementation of LolliMon to achieve reasonable performance while verifying the shape of data structures.

**Heap Context and Mode Analysis** When LolliMon is called to prove an assertion, the logical context contains the programmer defined clauses and the logical encoding of the program heap. Naively, we can traverse the heap and dump out all the contents into the logical context before we start the proof. However, such an approach will never work in practice. For one thing, we are doubling the memory requirements to use any assertion, even one that is related to only a single location in the program's heap. For another, the performance of LolliMon will suffer from a large program heap. As we mentioned earlier, LolliMon uses the Tag Frame Fast system, a very efficient system to deal with linear resources, but the rule for picking a certain formula in the context to prove the goal still takes time that is linear to the number of formulas in the context. This means that the larger the program heap, the larger the logical context, and the worse the performance.

Fortunately, it is not necessary to dump the formulas describing the heap into the context. Since the formulas describing memory locations are all of the form  $(\text{lin } \ell v)$  where  $\ell$  is the address of the location and  $v$  is its contents, we can use a hash table with the addresses of the locations as keys to manage the formula tags. To determine the value stored in a location, we will simply look it up in the program's native heap. The context for the memory contents therefore consists of the program heap itself plus the hash table. This hash table speeds up the proof search in two ways. One, it takes amortized constant time to look up the predicate  $(\text{lin } \ell v)$  in the context. Two, it separates the heap formulas from the rest of the formulas in the context, and therefore greatly decrease the time needed to pick a formula

<sup>3</sup>LolliMon also gives forward-chaining semantics to the synchronous connectives. However, we have not used this feature in our examples so far.

in the context. Furthermore, we exploit the tagging system so that we only need to create bindings for locations that have directly been used in proving the goal in the hash table. Consequently, the memory overhead of the hash table for tags is linear to the size of the data structure that is of interest to the assertion,

In order for this optimization to be correct, we assume that we never put  $(\text{lin } \ell \ v)$  in negative positions, which means that we do not add new  $(\text{lin } \ell \ v)$  predicates into the context as the proof search goes. If we added new bindings during the proof, it would not be sound to only refer to the heap when determining the contents of a location. We also must enforce that whenever we attempt to prove goal  $(\text{lin } \ell \ v)$ , term  $\ell$  is ground (already known). Otherwise, the hash table is of no use. These assumptions so far have not restricted the examples that we can check. Intuitively, we want to look up the contents of specific memory locations, but we never need to ask the question of which location contains a specific value.

The first assumption can be easily checked by syntactically traversing the structure of the goal and the clauses. The second assumption can be checked using LolliMon’s mode analysis. The basic idea of mode analysis is to declare the *input* and *output* modes for the arguments of each predicate. The arguments with *input* mode have to be ground before proving the predicate, and the arguments with *output* mode have to be ground when the predicate is proved. Mode analysis in logic programming languages checks the information flow of each clause definition to determine if this definition obeys the modes declaration. Because we would like to make sure that the first argument of the  $(\text{lin } \ell \ v)$  predicate is always ground when we try to prove it, we declare its mode as `#mode lin +L -V`. We declare the modes of other predicates similarly. Finally, we check that the formula to be asserted is also well-moded. Mode analysis guarantees us that predicate  $(\text{lin } \ell \ v)$  will always have a ground term  $\ell$  when it needs to be proven.

## 5. EXPRESSING ALIASING

All the examples we have shown up to this point do not have aliased data structures. Although linear logic is extremely well-suited for reasoning about disjoint memory locations, it can also express invariants involving aliasing. In this section, we show how to deal with aliasing by explaining two example linear logic programs for verifying the shape invariants of circular lists and DAGs.

### 5.1 Example: Circular Linked Lists

The logic program defining list predicate `llist` in Section 2.5 only succeeds on non-circular lists (as intended). It fails on circular lists because each list node has to be visited exactly once. To check for circularity, the first node in the list need to be visited again when the tail node is reached.

We can explicitly define circular linked lists by modifying the definition of `llist` slightly. The trick is to pass the address of the head node of the list as an argument to the circular list predicate. The program succeeds immediately when a node contains a pointer pointing back to the head node is reached, without attempting to follow that pointer. The logic program for circular lists is given below:

```
clistnode L T o- struct L (I1::T::nil);
                    ( struct L (I2::X::nil),
                      clistnode X T ).
clist L o- L is 0; clistnode L L.
```

Predicate `clistnode L T` means that the memory starting at location  $L$  is a list that eventually points to location  $T$ , which is the address of the head node of the list. Predicate `clist L` is the top-level predicate meaning that the memory starting from  $L$  is a circular linked list, and it checks that  $L$  is either a NULL pointer or that it is a list that eventually points back to itself.

For data structures that contain a small amount of specific aliasing, such as circular lists we can write predicates to carry the specific aliased locations around and make the program succeed immediately when these locations are encountered. In this way, we still visit each location exactly once.

### 5.2 Example: Directed Acyclic Graphs

A directed acyclic graph, or DAG, may contain aliased subgraphs. The trick from circular lists won’t work here, because we don’t have enough information about where the aliasing may occur. Instead we can use the `&` and `top` connectives to allow more flexibility for sharing between data structures.

Before going into the details of DAGs, we show a small example of how to use `&` and `top` in case of may aliasing. Recall that a memory is described by `F1 & F2` when it is described by both `F1` and `F2`, and that every memory can be described by `top`. Now we want to describe a memory that has two locations that may be aliased. Formula  $(\text{lin } L1 \ V1, \ \text{lin } L2 \ V2)$  can only describe memories that has exactly two unaliased locations. Here, we can use the following formula to describe this may-alias situation.

$$(\text{lin } L1 \ V1, \ \text{top}) \ \& \ (\text{lin } L2 \ V2, \ \text{top})$$

If the memory contains only one location  $\ell$ , then it can be described by both the sub-formulas connected by `&`. The tensor divides the memory into one part that  $\ell$  points to and the other part that is the empty memory;  $\ell$  is the witness for both  $L1$  and  $L2$  and `top` is satisfied by the empty memory. In the case where the memory contains two locations  $\ell$  and  $\ell'$ , the first sub-formula is satisfied by using  $\ell$  as the witness for  $L1$  and letting `top` consume the rest of the memory containing location  $\ell'$ ; similarly, the second sub-formula is satisfied by using  $\ell'$  as the witness for  $L2$  and letting `top` consume the rest of the memory containing location  $\ell$ .

When a DAG has no sharing between subgraphs, it becomes a tree. Let’s look at the definition of tree first, and then we will create the definition of a DAG by modifying the definition of a tree using the idea of sharing.

```
tree L o- (L is 0);
          (struct L (Data::Left::Right::nil),
            tree Left, tree Right).
```

The above tree definition states that each tree node contains data and a pointer to its right child and a pointer to its left child; that both of the children are trees; and that the tree node, the left subtree, and the right subtree are pair-wise disjoint.

We can modify the tree definition to describe a DAG by changing the tensor between the two subtrees to the additive



conjunction & so that they can be aliased. The definition of DAG is given below:

```
dag L :- (L is 0);
(struct L (Data::Left::Right::nil),
((dag Left, top) & (dag Right, top))).
```

The DAG definition still requires that the root node is disjoint from both subgraphs (so that there can be no cycles), but allows the two subgraphs to be aliased.

This definition may require a node to be checked multiple times. For example, consider the case where the DAG contains four nodes: A points to B and C, and both B and C point to D. In this case, proving `dag A` requires proving `dag B` and `dag C` using the same set of locations. Both of these subgoals will involve proving `dag D`. A node will be checked once for each unique path from the root of the dag to that node. However, the definition is still guaranteed to terminate since each use of the definition consumes the locations for one node.

## 6. EXTENDED EXAMPLE: RED-BLACK TREES

In this section, we will explore a longer example of red-black trees. Red black trees are balanced binary search trees that enforce various properties in order to guarantee basic operations take  $O(\lg n)$  time. In this example, we not only check the shape of the data structure, but also check other properties such as the partial ordering of the data carried at each node.

### 6.1 Expressing Red-Black Tree Invariants

Since red-black trees are binary search trees, the data at a left child is less than the data at the parent, and the data at the right child is greater than that at the parent.

The `checkData` predicate checks the relationship between the data `D` of the current node and the data `Pd` of the parent node. To do this it takes a flag `Rc` that states whether this node is a right child. If the node is a left child (`Rc` is 0), then the data must be less than or equal to that of the parent. If it is a right child (`Rc` is 1), then the data must be greater than or equal to the parent's data. In the special case of the root, which has no restriction on the data, `Rc` is set to 2.

```
checkData D Pd Rc o-
(Rc is 0, (D = Pd; Pd > D));
(Rc is 1, (D = Pd; D > Pd));
(Rc is 2).
```

A red node is a node that contains four elements: the color red (represented by 1), data, a pointer to a left child, and a pointer to a right child. It is a binary search tree node, so its data must be appropriately related to that of its parent. In a red-black tree, no red node may have a red parent, so both the left and right children must be black nodes. The black height of the two subtrees must be equal. The `rnode` predicate takes a location `L`, the parent data `Pd`, the flag `Rc`, and if `L` is a well-formed red node, returns the black height `Bh`.

```
rnode L Pd Rc Bh o-
(struct L (1::Data::Left::Right::nil),
(checkData Data Pd Rc,
bnode Left Data 0 Bh,
bnode Right Data 1 Bh).
```

All leaves in a red-black tree are black. A black node may be null (with black height of zero), or it may be an internal node, in which case it contains the same four elements as a red node, except that the color is black (represented by 0). Again, its data is correctly related to the parent's data. Black nodes may have either red children or black children, as long as they have the same black height. The black height of an internal black node is one greater than that of its children. The `bnode` predicate takes a location `L`, the parent data `Pd`, the flag `Rc`, and if `L` is a well-formed black node, returns the black height `Bh`.

```
bnode L Pd Rc Bh o-
(L is 0, Bh is 0);
(struct L (0::Data::Left::Right::nil),
checkData Data Pd Rc,
rbnode Left Data 0 Bh2,
rbnode Right Data 1 Bh2,
Bh is Bh2 + 1).
```

The `rbnode` predicate states that a node is red or black by checking that it is either a red node or it is a black node.

```
rbnode L Pd Rc Bh o-
(bnode L Pd Rc Bh); (rnode L Pd Rc Bh).
```

The `rbtree` node takes a location `L` and determines if it is the root of a red-black tree. The root of a red black tree must be black and has no restrictions on its data.

```
rbtree L o - bnode L 0 2 Bh.
```

In addition to these predicates, the definition of red-black tree invariants must also contain the type and mode for each predicate. For example

```
rnode: int -> int -> int -> int -> o.
#mode rnode +L +Pd +Rc -Bh.
```

### 6.2 A Red-Black Tree implementation

We took an implementation of red-black trees from The Object Oriented Programming Web (<http://www.oopweb.com>) and ported it to MiniC. Porting required the following changes. Type variables were changed to end in “\_tp”. Stack allocated structs were moved to the heap. `printf` was replaced with the simpler MiniC function `print` that takes a single expression. Variable declaration and initialization were broken down into two separate statements. Single line statement blocks were enclosed in curly braces.

We modified the LolloMon definitions given in the previous section to match this C implementation. We changed the ordering of the values in the struct and extended it to include a parent pointer and a key in order to correspond to the struct type declaration in the implementation. Instead of using null pointers to represent leaves, the implementation creates one node called `sentinel` to be used as a universal nil node. We modified the definition of our node predicates to take the address of this `sentinel` node as an extra argument. We pass in the location of the sentinel node as a second argument to the predicate `rbtree`. It determines that the sentinel is black, and then passes its address into each of the other predicates. The `bnode` predicate succeeds immediately when the address passed in is that of the sentinel, without attempting to check its structure.

```

bnode L N Pd Rc Bh o-
(L is N, Bh is 0);
(...as above...).

nilnode N o-
struct N (Left::Right::Parent::0
::Key::Data::nil).

rbtree Root Nilnode o-
nilnode Nilnode,
bnode Root Nilnode 0 2 Bh.

```

The main function in our implementation builds a red-black tree using a series of inserts, finds, and deletes. After each sequence of operations, it checks that the root variable still points to a red-black tree with sentinel variable as its nil node.

```

node_tp sentinel;
node_tp root;
...
assert([[rbtree $root $sentinel, top]]);

```

A subset of the code is available in Appendix B.

### 6.3 Finding Errors using Assertions

We did not find any errors in this implementation. However, we can introduce errors and show how they could be caught immediately using assertions.

When inserting a new node, the node is originally colored red. Then the function `insertFixup` checks to see if this has violated the invariant that no red node has a red parent. If so, a rotation is performed. Suppose we accidentally forget to call `insertFixup` and the new red node happen to be inserted under another red node, then the assertion that the current tree is a red black tree will fail because the `rnode` predicate requires its two children to be both black nodes, which cannot be proven by the current condition.

Our red-black tree invariants do not specifically deal with the parent pointers. However, parent pointers are used when determining which rotation to perform. A mistake in setting a parent pointer will not cause an assertion failure immediately, but will likely do so after an insert or delete causes a rotation in that part of the tree. Or if we chose, we could modify the invariants to ensure that the parent pointers are correctly aligned.

## 7. RELATED AND FUTURE WORK

In this section we discuss related work and future work.

### 7.1 Related Work

There has been a lot of research done in static shape analysis. Static analysis does not have runtime overhead. However, because of the lack of runtime information, static analysis usually leads to conservative and less precise results or requires many annotations or manual proofs. Among this work, Reynolds, O’Hearn et. al. developed separation logic [19] as an assertion language for verifying the correctness of pointer programs. Our idea of using a sub-structural logic to describe memory come directly from their work. The reasons why we chose to use intuitionistic linear logic instead of separation logic are that we do not need the bunched context to verify our examples and there is no theorem prover or

logic programming language for first-order separation logic to automatically discharge proof obligations.

Purify [7] is a tool that dynamically detects memory access errors and memory leaks. It traps every memory access calls in the program and use object code insertion technique to augment the programs with the dynamic checking logic. SWAT [8] is another tool that dynamically checks memory leaks. It uses a profiling infrastructure to monitor memory access operations and detect memory leaks, a simple uniform memory safety policy. Both of these works focus on dynamically detecting memory access errors and memory leaks, whereas our work mainly focus on dynamically checking complex programmer specified invariants about memory shapes.

Lastly, we use LolliMon [14] to prove the validity of the formula to be asserted. We modified the implementation of LolliMon by treating the linear context of clauses that are directly translated from the memory state as a hash table.

### 7.2 Future Work

In future work, we plan to design an assertion language that is easy for programmers to master. Right now, in order to use our system, a programmer must define clauses and write assertions in the syntax of LolliMon. Even though the logic programming language is declarative and relatively easy to learn, it still requires an added learning curve for programmers unfamiliar with logic programming. The goal of the assertion language design is to keep the declarative feature and at the same time bring the syntax of the assertion language closer to the syntax of defining data structures in the native language, so that the programmers have an easier time specifying invariants.

In this paper, we only implement a prototype system to check the feasibility of our basic idea of checking the invariants of recursive data structures dynamically using a linear logic programming language. A lot of work remains to make this system real. Ideally, we would like to deploy our system for ANSI C. The questions to be solved include how to link the runtime system of C with the logic engine, how the performance will scale to large data structures, and how to optimize the logic engine.

We have looked into statically verifying properties of the programs using linear logic [3, 12, 13]. In the long term, we hope to explore how to interface the dynamic system with the static verification system.

*Acknowledgments.* We would like to thank David Walker for meaningful discussions and constructive comments. This research was supported in part by ARDA Grant no. NBCHC-030106, National Science Foundation grants CCR-0238328 and CCR-0208601 and an Alfred P. Sloan Fellowship. This work does not necessarily reflect the opinions or policy of the federal government or Sloan foundation and no official endorsement should be inferred.

## 8. REFERENCES

- [1] Programming with assertions. <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>.
- [2] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science*, pages 33–44, Ottawa, Canada, June 2003.

- [3] A. Ahmed and D. Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New Orleans, Jan. 2003.
- [4] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Programming Languages and Systems*, 23(5):657–683, 2001.
- [5] L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *ACM Symposium on Principles of Programming Languages*, pages 220–231, Venice, Italy, Jan. 2004.
- [6] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
- [7] R. Hastings and B. Joyce. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136. USENIX Association, 1992.
- [8] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.
- [9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.
- [10] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *Papers presented at the IEEE symposium on Logic in computer science*, pages 327–365, Orlando, FL, USA, 1994. Academic Press, Inc.
- [11] S. Ishtiaq and P. O’Hearn. BI as an assertion language for mutable data structures. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, Jan. 2001.
- [12] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In P. Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 407–416. IEEE Computer Society Press, June 2005.
- [13] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. Technical Report TR-738-05, Department of Computer Science, University of Princeton, 2005.
- [14] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 35–46, New York, NY, USA, 2005. ACM Press.
- [15] P. López and J. Polakow. Implementing efficient resource management for linear logic programming. In *LPAR*, pages 528–543, 2004.
- [16] B. Meyer. Eiffel: programming for reusability and extensibility. *SIGPLAN Not.*, 22(2):85–94, 1987.
- [17] G. Morrisett, A. Ahmed, and M. Fluet. L<sup>3</sup>: A linear language with locations. In *Seventh International Conference on Typed Lambda Calculi and Applications*, 2005.
- [18] P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, number 2142 in LNCS, pages 1–19, Paris, 2001.
- [19] 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. IEEE Computer Society, 2002.

## APPENDIX

### A. SEQUENT CALCULUS FOR INTUITIONISTIC LINEAR LOGIC

$$\boxed{\Gamma; \Delta \longrightarrow F}$$

$$\frac{}{\Gamma; F \longrightarrow F} \text{L-Init} \quad \frac{\Gamma, F; \Delta, F \longrightarrow F'}{\Gamma, F; \Delta \longrightarrow F'} \text{Copy}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow F_1 \quad \Gamma; \Delta_2 \longrightarrow F_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow F_1 \otimes F_2} \otimes R \quad \frac{\Gamma; \Delta, F_1, F_2 \longrightarrow F}{\Gamma; \Delta, F_1 \otimes F_2 \longrightarrow F} \otimes L$$

$$\frac{\Gamma; \Delta, F_1 \longrightarrow F_2}{\Gamma; \Delta \longrightarrow F_1 \multimap F_2} \multimap R \quad \frac{\Gamma; \Delta \longrightarrow F_1 \quad \Gamma; \Delta', F_2 \longrightarrow F}{\Gamma; \Delta, \Delta', F_1 \multimap F_2 \longrightarrow F} \multimap L$$

$$\frac{}{\Gamma; \cdot \longrightarrow \mathbf{1}} \mathbf{1}R \quad \frac{\Gamma; \Delta \longrightarrow F}{\Gamma; \Delta, \mathbf{1} \longrightarrow F} \mathbf{1}L$$

$$\frac{\Gamma; \Delta \longrightarrow F_1 \quad \Gamma; \Delta \longrightarrow F_2}{\Gamma; \Delta \longrightarrow F_1 \& F_2} \&R$$

$$\frac{\Gamma; \Delta, F_1 \longrightarrow F}{\Gamma; \Delta, F_1 \& F_2 \longrightarrow F} \&L1 \quad \frac{\Gamma; \Delta, F_2 \longrightarrow F}{\Gamma; \Delta, F_1 \& F_2 \longrightarrow F} \&L2$$

$$\frac{}{\Gamma; \Delta \longrightarrow \top} \top R$$

$$\frac{\Gamma; \Delta \longrightarrow F_1}{\Gamma; \Delta \longrightarrow F_1 \oplus F_2} \oplus R1 \quad \frac{\Gamma; \Delta \longrightarrow F_2}{\Gamma; \Delta \longrightarrow F_1 \oplus F_2} \oplus R2$$

$$\frac{\Gamma; \Delta, F_1 \longrightarrow F \quad \Gamma; \Delta, F_2 \longrightarrow F}{\Gamma; \Delta, F_1 \oplus F_2 \longrightarrow F} \oplus L$$

$$\frac{}{\Gamma; \Delta, \mathbf{0} \longrightarrow F} \mathbf{0}L$$

$$\frac{\Gamma; \Delta \longrightarrow F[t/x]}{\Gamma; \Delta \longrightarrow \exists x.F} \exists R \quad \frac{\Gamma; \Delta, F[a/x] \longrightarrow F'}{\Gamma; \Delta, \exists x.F \longrightarrow F'} \exists L$$

$$\frac{\Gamma; \Delta \longrightarrow F[a/x]}{\Gamma; \Delta \longrightarrow \forall x.F} \forall R \quad \frac{\Gamma; \Delta, F[t/x] \longrightarrow F'}{\Gamma; \Delta, \forall x.F \longrightarrow F'} \forall L$$

$$\frac{\Gamma; \cdot \longrightarrow F}{\Gamma; \cdot \longrightarrow !F} !R \quad \frac{\Gamma; F; \Delta \longrightarrow F'}{\Gamma; \Delta, !F \longrightarrow F'} !L$$

### B. REDBLACKTREE.MINIC

Below we include partial code for the MiniC implementation of red-black trees<sup>4</sup>. The delete function includes an assertion to verify that deleting a node has not violated the red-black tree invariants defined at the beginning of the program.

```
/* -----
 * Red-Black Tree invariants specified
 * as LolliMon predicates
 * ----- */
```

```
[[
checkData: int -> int -> int -> o.
```

<sup>4</sup>based on code from [http://oopweb.com/Algorithms/Documents/Sman/Volume/s\\_rbt.txt](http://oopweb.com/Algorithms/Documents/Sman/Volume/s_rbt.txt)

```

bnode: int -> int -> int -> int -> int -> o.
rnode: int -> int -> int -> int -> int -> o.
rbnode: int -> int -> int -> int -> int -> o.
rbtree: int -> int -> o.
nilnode: int -> o.

```

```

#mode checkData +X +Y +Z.
#mode rnode +L +N +P +G -B.
#mode bnode +L +N +P +G -B.
#mode rbnode +L +N +P +G -B.
#mode rbtree +L +N.
#mode nilnode +L.

```

```

checkData D Pd Rc o-
  (Rc is 0, (D = Pd; Pd > D));
  (Rc is 1, (D = Pd; D > Pd));
  (Rc is 2).

```

```

rnode L N Pd Rc Bh o-
  (struct L (Left::Right::Parent
            ::1::Key::Data::nil),
   checkData Data Pd Rc,
   bnode Left N Data 0 Bh,
   bnode Right N Data 1 Bh).

```

```

bnode L N Pd Rc Bh o-
  (L is N, Bh is 0);
  (struct L (Left::Right::Parent
            ::0::Key::Data::nil),
   checkData Data Pd Rc,
   rbnode Left N Data 0 Bh2,
   rbnode Right N Data 1 Bh2,
   Bh is Bh2 + 1).

```

```

rbnode L N Pd Rc Bh o-
  (bnode L N Pd Rc Bh);
  (rnode L N Pd Rc Bh).

```

```

nilnode N o-
  struct N (Left::Right::Parent
            ::0::Key::Data::nil).

```

```

rbtree Root Nilnode o-
  nilnode Nilnode,
  bnode Root Nilnode 0 2 Bh.
]]

```

```

/* -----
 * Declaration of node type
 * ----- */

```

```

/* Red-Black tree description */
enum nodecolor_tp {BLACK, RED};

```

```

/* node type */
struct nodeTag_tp {
  struct nodeTag_tp *left;
  struct nodeTag_tp *right;
  struct nodeTag_tp *parent;
  nodecolor_tp color;
  key_tp key;
  rec_tp rec;
};
typedef struct nodeTag_tp node_tp;

```

```

/* -----
 * Global variables: sentinel and root
 * ----- */

```

```

/* all leafs are sentinels */

```

```

node_tp *sentinel;

```

```

/* root of Red-Black tree */
node_tp *root;

```

```

/* -----
 * Function Declarations
 * (code omitted)
 * ----- */

```

```

void rotateLeft(node_tp *x) {...}
void rotateRight(node_tp *x) {...}
status_tp find(key_tp key, rec_tp *rec) {...}
void insertFixup(node_tp *x) {...}
status_tp insert(key_tp key, rec_tp *rec) {...}
void deleteFixup(node_tp *x) {...}

```

```

/* delete node z from tree */
status_tp delete(key_tp key) {

```

```

  node_tp *y;
  ...code omitted...
  free(y);

```

```

/* assert that delete maintains the invariants */
assert([[rbtree $root $sentinel, top]]);

```

```

  return STATUS_OK;
}

```

```

/* -----
 * Main Function
 * ----- */

```

```

int main () {

```

```

  rec_tp *rec;
  status_tp status;
  int i;

```

```

/* build sentinel (nil) node */
sentinel = malloc(sizeof(node_tp));
sentinel->left = sentinel;
sentinel->right = sentinel;
sentinel->color = BLACK;
sentinel->key = 0;
sentinel->rec = 0;

```

```

/* allocate record */
rec = malloc(sizeof(rec_tp));

```

```

/* assign initial value of root */
root = sentinel;

```

```

/* fill in with keys 0 through 14 */
i = 0;
while (i < 15) {
  rec->stuff = i + 20;
  status = insert(i,rec);
  i = i + 1;
}

```

```

/* delete nodes */
status = delete(3);
}

```