Expressing Heap-shape Contracts in Linear Logic *

Frances Perry Limin Jia David Walker
Princeton University
{frances, ljia, dpw}@cs.princeton.edu

Abstract

Contracts (dynamically checked programmer assertions) are a widely accepted mechanism for specifying, checking and documenting properties of software components. Most, if not all, contract systems expect programmers to use the native programming language to express their program invariants. While this is most effective for many simple invariants, expressing properties of data structures and aliasing patterns can be extremely complicated. If written in the native language in an unstructured way, such contracts are bound to be unclear and ineffective as documentation. In this paper, we show how to use linear logic as a language of contracts for an imperative programming language. The high-level nature of our linear logical contracts makes specifying memory shape and aliasing properties of complex recursive data structures easy. Moreover, since we give our logic a clear, compositional semantics, the contracts serve as effective, executable documentation for programmer expectations. In order to evaluate the truth of our linear logical contracts at run time, we use a modified version of LolliMon, a linear logic programming language.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Reasoning about Programs

General Terms Languages, Reliability, Theory, Verification

Keywords Contracts, Assertions, Heap Shape, Linear Logic

1. Introduction

The recent research on separation logic by O'Hearn, Reynolds, Yang, et al. [19, 9, 22] 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:

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

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 correctness 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 [3]. However, this proof was done by hand, and manual verification is clearly far too heavyweight for ordinary programmers to incorporate into their everyday programming or debugging tasks. Happily, there is much current research focused on automating the process, but it is nevertheless likely to be decades before ordinary programmers are able to use it productively.

On the other hand, people have studied and used *contracts* since the 1970s [20, 8, 14, 16] Contracts are specifications that programmers use to document component requirements and to clearly express guarantees. Importantly, contracts are *executable* specifications that are evaluated at run time and actually enforce the specified properties. When used consistently, contracts can help improve the clarity of code and detect programmer errors. All kinds of past and current languages include these features, including Eiffel [15], Java [21] and Scheme [4].

Most contract systems expect programmers to use the native programming language to express their program invariants. While this technique is most effective for many simple invariants, expressing properties of data structures and aliasing patterns can be extremely complicated. In fact, any naive "roll your own" function a programmer might write to check heap-shape properties would have to set up substantial infrastructure to keep track of sets of locations and to record and check aliasing properties. If this infrastructure is set up in an ad hoc, verbose, and unstructured manner, the meaning of contracts will be unclear and their value as documentation substantially diminished.

Hence, in this paper, we seize upon the critical insight of O'Hearn, Reynolds, Yang and others: substructural logics, separation logic being a special case, can be a highly effective specification mechanisms for properties of heap shape. However, rather than attempting to verify that substructural specifications hold *statically*, we explore using them as a language of contracts, evaluated *dynamically*. When used as contracts, substructural specifications

* This research was supported in part by NSF grant CCR-0238328 and an Alfred P. Sloan Fellowship. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the NSF or Sloan foundation and no official endorsement should be inferred.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

are much lighter-weight verification mechanisms than when used in static verification. A programmer can simply plop down a contract wherever they choose in their program, employing a pay-as-you-go strategy for making their programs more reliable. Consequently, using substructural logics as contracts provides an opportunity for a much more immediate impact on programming practice. Of course, using logical assertions as contracts does not rule out using them as annotations for static verification as well. On the contrary, the two are quite complementary. If a programmer wishes to statically verify a small portion of a large program, they may do so soundly only if they are able check the preconditions of their program fragment dynamically when control flows into the fragment. The most effective way to engineer this is to ensure that one can interpret any assertion *both* as part of a static program analysis *and* as a legal contract that may be evaluated dynamically. In this paper, we focus exclusively on evaluation of substructural assertions dynamically, but we are well aware of progress on static verification using these same assertions and hope to exploit that in the future.

To be more specific, we have implemented an imperative programming language in which users can specify heap-shape contracts using a fragment of linear logic. Linear logic shares a great deal in common with the logic of bunched implications [18], upon which separation logic is based. In particular, linear logic's multiplicative connectives can specify separation or disjointedness of heap state while its additive connectives can specify sharing. Linear logic's unrestricted modality allows us to include and repeatedly reuse specifications that are independent of the current heap shape. The main motivation for using linear logic as opposed to separation logic is that to enforce contracts, we require some mechanism for *evaluating* logical specifications at run time. To evaluate contracts, we incorporate a modified version of Lopez et al.'s linear logic engine LolliMon [12], a decedent of Lolli [7], into the run-time system of our language.

In summary, our main contributions are:

- The *idea* that a substructural logic can be used as a language of contracts for specifying heap-shape properties. Unlike the ad hoc, unstructured, heap-shape contracts one might write in native code, these contracts serve as clear, compact and semantically well-defined documentation of heap-shape properties.
- An implementation of this idea using linear logic as the logical specification language for heap shapes. Contracts written in linear logic are evaluated using a modified version of LolliMon, and used to verify properties of data structures coded in MiniC, which, despite its name, is a substantial fragment of the C programming language.
- An implemented collection of example contracts that show the effectiveness of our technique including singly linked lists, circular lists, doubly linked lists, trees, directed acyclic graphs, and other more complicated data structures such as red-black trees and b-trees.
- An indexed model of linear logic augmented with inductive definitions and a proof of soundness of our implementation technique.

The rest of the paper is organized as follows. In Section 2 we give an informal overview of our MiniC system and its linear logical contract language. In Section 3, we introduce an indexed memory model for our logic and prove the soundness of our technique. In Section 4, we provide implementation details of the MiniC Language and its interface with LolliMon. In Section 5 we discuss some more complex data structures, including those involving aliasing. In Section 6, we explore example code for red-black trees. And finally, in Section 7 we discuss related work and some directions for future research. Please note a preliminary version of this paper was presented at the SPACE 06 workshop, January 2006. However, that workshop has no formal, published proceedings.

2. Overview

In this section, we illustrate the basic components of our system by going over a simple program that includes a call to an untrusted `copy` function. We also give an example of how to verify recursively defined shape invariants of singly linked lists.

2.1 A Simple MiniC Program

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 describing memory shapes of concern. We will explain them in the next subsection.

In the `main` function, the programmer allocates a struct of type `pair_tp`, and 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 would like to check that 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 describes the desired shape of the memory.

2.2 Specifying Invariants in LolliMon

We use LolliMon [12], a monadic concurrent linear logic programming language, as our assertion language. We have not used the monadic features of LolliMon in our examples, so we omit them from the paper.

In LolliMon, we 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. Following the style of Prolog, we write clauses as inverted implication: `Head o- Body`. 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. Logic variables are capitalized. Below is a sample program in LolliMon.

```
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)).
```

In lines 1 through 3, we defined two clauses for predicate `struct` that together describe a memory chunk with starting address `L` and list of values stored `X`. We use predicate (`lin L V`) to describe a memory cell whose address is `L` and contents is `V`. The multiplicative conjunction, written as "," in LolliMon, enforces that the locations in its subformulas are disjoint. For example, there is no memory that satisfies the formula $((\mathsf{lin}\ 10\ 100), (\mathsf{lin}\ 10\ 100))$, because that would require location 10 to be in two disjoint pieces of memory. 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`)[1].

[1] Notice that locations are treated at a high level of abstraction. Adjacent fields in a struct or array are offset by 1.

```
[[
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

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 query. The last argument is the goal formula to be proved.

Here 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 at location 10 and contains two elements `X` and `Y`, and the second piece of memory starts at 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.

2.3 Verifying a Simple MiniC Program

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. These assertions may include program variables by prefixing them with a dollar sign. Intuitively, at run time when an assert is reached, the clause definitions and the formula describing the current program memory are given as the logical context to the LolliMon engine. The formula to be asserted is sent to the engine as the query formula to be executed against the context. If the query is proven by the logic engine, the program will continue running; if it fails, the program will be aborted.

Assume the `copy` function called by the code in Figure 1 has a correct implementation:

```
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` (allocated by `main`) and `pair2` (allocated by `copy`). Assuming `pair1` is allocated at location `L1` and `pair1` is allocated at location `L2`, then the formula describing the current heap is

$$(\texttt{lin}\ \texttt{L1}\ 100),\ (\texttt{lin}\ (\texttt{L1}+1)\ 200),\\ (\texttt{lin}\ \texttt{L2}\ 100),\ (\texttt{lin}\ (\texttt{L2}+1)\ 200)$$

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

```
struct L1 (X::Y::nil),  struct L2 (X::Y::nil)
```

The above formula is checked against the clauses defined in the first three lines of `pair.minic` and the formula describing the current 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:

```
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.

2.4 Specifying the Shape of Linked Lists

Here we show how to describe the invariants of non-circular singly linked lists. Predicate (`llist` `L`) means that the memory chunk starting from location L is a singly linked list. A location `L` points to a list if either it is null (0) or it contains data `Data` and the value `Next` in the adjacent location is a pointer to a list. Additive disjunction is written as ";" in LolliMon. Formula $(F_1; F_2)$ describes a memory that can be described by either F_1 or F_2. For example, the memory that contains just the location 10 and its contents 100 can satisfy $((\text{lin } 10\ 100); (\text{lin } 10\ 200))$ because only one of the subformulas need to be satisfied.

```
llist L o- (L is 0);
           (struct L (Data::Next::nil),
            llist Next).
```

In the MiniC program `llist.minic` in Figure 2, the LolliMon definitions are between line 1 and 9. In addition to the above clause definition, the LolliMon definitions also include the type declaration of the `llist` predicate (line 2) which specifies that the `llist` predicate takes one argument of type int (a fully applied predicate always has the special object type o); and the mode declaration (line 4), which will be explained in Section 4. Following the LolliMon definitions are the MiniC definitions that declare a list node, `struct node_tp` (line 11–15).

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

4  #mode llist +L.

6  llist L o- (L is 0);
7             (struct L (Data::Next::nil),
8              llist Next).
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([[llist $list1]]);

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

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

40    return 0;
41  }
```

Figure 2. llist.minic

In lines 20 through 29, the `main` function constructs a list matching the one in Figure 3. Assume that `list3` is allocated at location ℓ_3, `list2` at ℓ_2, and `list1` at ℓ_1. The programmer then asserts at line 32 that `list1` is a linked list (`[[llist $list1]]`). After replacing the variable with its value, the assertion formula becomes (`llist` ℓ_1). Since ℓ_1 is not 0, the logic engine needs to solve the subgoal on line 7 and 8. The linear resources corresponding to memory m_1 in Figure 3 are used to determine that `Data` is 1 and `Next` is ℓ_2. The logic engine then attempts to prove that ℓ_2 is a pointer to a list using the remainder of the memory m_2 and m_3. This in turn reduces to proving that ℓ_3 points to a list using m_3. The

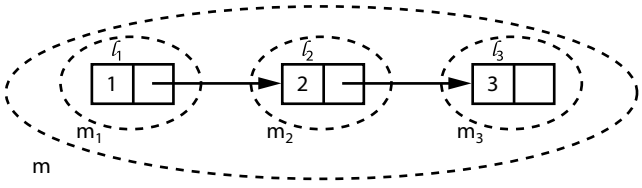

Figure 3. Memory containing a linked list.

```
% ./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.
Assertion [[llist $list1]] Failed
     at Position 38.4 - 38.28
```

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

base case is reached when proving that 0 is a pointer to 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 assertion, at line 38, begins much like the first. However, when the subgoal of proving ℓ_3 points to a list is reached, the logic engine tries to use the resources corresponding to m_3 to prove that ℓ_3 points to a list. The next step would be to prove that ℓ_1 points a list using no linear resource. Unlike the first assert, where we can prove that 0 is a list pointer without using any linear resources, proving that ℓ_1 points to 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 fails, and the MiniC interpreter throws an `Assert Failed` exception.

The actual output from running `llist.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 query with the program variables substituted with the actual values. And finally, the result of running the query is printed.

3. Memory Semantics of Linear Logic

In this section, we begin by reviewing the formal syntax of linear logic and explaining the formal syntax of recursively defined predicates. Next we introduce an indexed semantics of the intuitionistic linear logic with recursive definitions. Finally, we present our formal result: 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 Formal Syntax

The syntax of the formulas is standard, and it includes both the multiplicative and additive linear connectives as well as the unrestricted modality !.

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

We use ℓ 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 (ℓ+1) for the address immediately after ℓ. The basic predicates include user defined predicates, arithmetic predicates, and ($\mathsf{lin}\ \ell\ v$).

Predicate ($\mathsf{lin}\ \ell\ v$) describes a memory that contains only one location ℓ with the contents v. The multiplicative connective $\otimes$ (similar to * in separation logic) separates the memory into two disjoint pieces. For example, memory m in Figure 3 can be described by (`struct` ℓ_1 (`1` :: ℓ_2 :: `nil`)) $\otimes$ (`struct` (ℓ_2 (`2` :: ℓ_3 ::

nil)) $\otimes$ (struct ℓ_3 (3 :: 0 :: nil)). The connective **1** is the unit of 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. Memory m_1 in Figure 3 can be described by (llist $\ell_2 \multimap$ llist ℓ_1). The connective $\top$ is the unit of the additive conjunction and describes all memories. The additive conjunction $F_1 \& F_2$ describes memories that satisfy both F_1 and F_2. For example, any memory m that satisfies F also satisfies $(F \& \top)$. Formula $F_1 \oplus F_2$ describes memories that satisfy either F_1 or F_2. Connective **0** is 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 relationships over integer expressions. For example, $(x = 2) \oplus (2 < x)$ is true if x is greater than or equal to two.

3.2 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 recursively defined predicate, and *Is* to denote the list of such definitions.

$$\begin{array}{llll} \textit{Pred def} & I & ::= & \forall x_1 .... \forall x_m . (F \multimap P\ x_1 \ldots x_n) \\ \textit{Pred defs} & Is & ::= & \cdot \mid I, Is \end{array}$$

Each definition I corresponds to a clause definition in LolliMon with the free logical variables universally quantified. Predicate $(P\ x_1 \ldots x_n)$ 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 $(P\ x_1 \ldots x_n)$. For example, below is the clause definition for llist in Section 2.4 given in the form of *I*:

$$\begin{array}{ll} \forall l.\ \forall d.\ \forall n. & (\ (l = 0) \\ & \quad \oplus ((\texttt{struct}\ l\ (d :: n :: nil)) \otimes (\mathsf{llist}\ n))\ ) \\ & \multimap \mathsf{llist}\ l \end{array}$$

3.3 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 [22]. The indexing scheme is inspired by the indexed semantics model for recursive types developed by Appel et al. [2] and the indexed memory model in recent work by Morrisett et al. [17]. A memory m maps locations ℓ to values v. We write $dom(m)$ to denote the set of locations in the domain of m and $m(\ell)$ to denote the value stored in ℓ. 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 $m \vDash^n_{Is} F$ states that given the set of predicate definitions Is, memory m can be described by formula F with n steps of approximation. The semantics for most of the linear logic connectives are straightforward, and they are not affected by the indexing scheme. In these cases, the index number n is just carried through the semantic judgments. The interesting case where the index number changes is the semantics for recursively defined predicates such as the struct and llist predicates from previous examples. The semantics of formulas is given in Figure 5.

For the semantics of recursively defined predicates, 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 $(P\ t_1 ... t_n)$ at the n^{th} unrolling, when m satisfies the clause body at

$\boxed{m \vDash^n_{Is} F}$

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

Figure 5. Semantics of Formulas

the $(n-1)^{th}$ unrolling. A predicate unrolls to $F[t_1/x_1]...[t_n/x_n]$ when $(\forall x_1 .... \forall x_m . (F \multimap P\ x_1 ... x_n)) \in Is$.

Now we use the semantics of a list predicate ($\texttt{llist}'\ \ell$) 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:

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

We use S_n to represent the set of memories that satisfy formula ($\texttt{llist}'\ \ell$) at the n^{th} approximation ($S_n = \{m | m \vDash^n_{Is} \texttt{llist}'\ \ell\}$).

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

When the index n is 0, we have the least precise idea of what the memories that satisfy $\texttt{llist}'\ \ell$ 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 ($\texttt{llist}'\ \ell$) at 0^{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 semantic judgment becomes more precise. As the index n approaches positive infinity, we reach the fixed point.

Another case worth discussing is the semantic judgment of linear implication $m \vDash^n_{Is} F_1 \multimap F_2$. 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 m' satisfying F_1, satisfies F_2.

The following lemma states that the n^{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 3.5.

Lemma 1 (Downward Closure)
For all $n \geq 1$, if $m \vDash^n_{Is} F$ then for all j, $0 \leq j < n$, $m \vDash^j_{Is} F$.

3.4 Soundness of Logical Deduction

The sequent calculus of linear logic is of the form: $\Gamma; \Delta \longrightarrow F$. Context Γ contains unrestricted resources, and context Δ 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. A memory m is described by the unrestricted context Γ and the linear context Δ if m is described by the formula created by wrapping each formula in Γ with ! and then tensoring together these formulas and all the formulas in Δ.

We prove that if memory m is described by contexts Γ and Δ, then m is also described by the logical consequence of Γ and Δ.

Theorem 2 (Soundness of Logical Deduction)
If $\Gamma; \Delta \longrightarrow F$ and for all $n \geq 0$, $m \vDash^n_{Is} \Gamma; \Delta$ implies $m \vDash^n_{Is} F$.

3.5 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 Γ. The formulas describing each allocated location in the current program heap are dumped into the linear context Δ. We use the notation $\mathsf{Locs}(m)$ to represent the set of formulas created by encoding each live heap location ℓ containing value v into its describing formula $(\mathsf{lin}\ \ell\ v)$.

We first show that the recursive definitions Is are valid with an empty memory.

Lemma 3
For all $n \geq 1$, $\cdot \vDash^n_{Is} !Is$

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

Lemma 4
For all $n \geq 0$, $m \vDash^n_{Is} \bigotimes(\mathsf{Locs}(m))$

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 the unrestricted context built from the definitions of the recursively defined predicates and the linear context built from 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)
If $Is; \mathsf{Locs}(m) \longrightarrow F$, then $\forall n \geq 1, m \vDash^n_{Is} F$.

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 LolliMon clause definitions in LolliMon, which are enclosed in double square brackets. The implementation automatically includes the definitions of the basic predicates such as `lin` and `struct`. Next there is a sequence of top level declarations that can include global variables, struct and union definitions, type definitions, function declarations, and enumerations. Program execution begins with the distinguished `main` function.

In an `assert` statement, the formula to be asserted is in double square brackets. Program variables may be included in the formula by prefixing them with a dollar sign.

The MiniC interpreter is written in OCaml. It is completely standard, except for the interpretation of `assert` statements. When the interpreter encounters an `assert` statement, 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; otherwise, the interpreter halts with an `Assertion Failed` exception.

4.2 The Logic Engine

We use LolliMon [12] as the logical engine 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.

Because linear logic requires that the formulas in the linear context have to be used exactly once, the resource management for a linear logic programming language can be quite complicated. The resource management of LolliMon implements the Tag Frame Fast System [13]. 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 manages to make most context manipulating operations take constant time, except the `pick` rule, which goes through the context linearly to choose a formula 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, the resources management of LolliMon contains the `pick` rule taking 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 those formulas are all of the form $(\mathsf{lin}\ \ell\ v)$ where ℓ 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 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 $(\mathsf{lin}\ \ell\ v)$ in the context. Two, it separates the heap formulas from the rest of the formulas in the context, and therefore greatly decreases the time needed to pick a formula 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 $(\mathsf{lin}\ \ell\ v)$ in negative positions, which means that we do not add new $(\mathsf{lin}\ \ell\ v)$ 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 $(\mathsf{lin}\ \ell\ v)$, term ℓ 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 $(\mathsf{lin}\ \ell\ v)$ 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.

5. Further Examples: 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 two examples of how to express invariants of aliased data structures using our logic.

5.1 Circular Linked Lists

The logic program defining the list predicate `llist` in Section 2.4 only succeeds on non-circular lists (as intended). It fails on circular lists because each list node can only be visited once. To check for circularity, the first node in the list need to be visited again when the tail node is reached.

We can define circular linked lists by modifying the definition of `llist` to pass around the address of the head node. The program succeeds immediately when a node containing a pointer back to the head node is reached, without attempting to follow that pointer.

```
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` states that location `L` is a list that eventually points to location `T`, the address of the head node of the list. Predicate `clist L` is the top-level predicate for a circular list that checks that either `L` is 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 that carry the specific aliased locations around and succeed immediately when these locations are encountered. In this way, we still visit each location exactly once.

5.2 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 use additive conjunction (`&`) and the `top` connective to allow sharing between data structures. Remember that formula `F1 & F2` describes memories that satisfy both `F1` and `F2`. The connective `top` describes all memories.

Before going into the details of DAGs, we show a small example of how to express that a pair of locations `L1` and `L2` may or may not be aliased. Formula (`lin L1 V1, lin L2 V2`) fails to describe such memories, because it only describes memories with two unaliased locations. Instead, we can use the following formula to describe the may-alias situation.

```
(lin L1 V1, top) & (lin L2 V2, top)
```

If the memory contains only one location ℓ, then it can be described by both the sub-formulas connected by &. The tensor divides the memory into one part containing ℓ and one part containing the empty memory; ℓ 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 ℓ and ℓ', the first sub-formula is satisfied by using ℓ as the witness for L1 and letting `top` consume the rest of the memory containing location ℓ'; the second sub-formula is satisfied by using ℓ' as the witness for L2 and letting `top` consume the rest of the memory containing location ℓ.

When a DAG has no sharing between subgraphs, it becomes a tree. Now we examine the definition for trees, which is given below. It states that each tree node contains data, 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 pairwise disjoint.

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

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.

```
dag L o- (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 the same node to 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 memory for the root node.

6. Extended Example: Red-Black Trees

In this section, we will explore an example of red-black trees, which are balanced binary search trees. We not only check the shape of the data structure, but also check the partial ordering of the data carried at each node, that red nodes do not have red parents, and that the black heights of all leaves are equal.

6.1 Expressing Red-Black Tree Invariants

First, we define an auxiliary predicate, `checkData`, to describe the ordering between the data `D` of the current node and the data `Pd` of the parent node. It also takes a flag `Rc` to indicate if this node is a right child (`Rc` is 1), left child (`Rc` is 0), or the special case, root (`Rc` is 2), where there is no restriction on the data.

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

Next we define predicate `rnode L Pd Rc Bh` to mean that the starting address of this red node is `L`, and the black height of all the leaves under this node is `Bh` (arguments `Pd` and `Rc` have the same meaning as above). Similarly, predicates `bnode` and `rbnode` describe black nodes and nodes of either color respectively. The definitions are given below. A red node contains four elements: the color red (represented by 1), data, pointers to a left child and a right child. The data must be appropriately related to that of its parent, both the left and right children must be black nodes, and the black height of the two subtrees is equal.

```
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).

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).

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

A black node is similar to a red node, except that the children may be either color, the black height increases by one, and an additional case to handle leaf nodes.

Finally, location `L` is a pointer to a red-black tree (`rbtree L`) if it is a black node of some black height.

```
rbtree L o-  bnode L 0 2 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. The porting process was very simple, and all changes were due to the simplicity of the MiniC parser and interpreter.

We modified the LolliMon definitions given in the previous section to include a parent pointer and a key in the `struct` predicate in order to match this C implementation. In the implementation, leaf nodes point to a universal nil node called `sentinel` node. The `sentinel` is heavily aliased and we applied the trick in the circular linked list example and modified the node predicate definitions to take the address of this `sentinel` node as an extra argument.

We inserted assertions in the main function after each series of operations on the red-black tree to check that the root variable still pointed to a red-black tree with sentinel variable as its nil node. A subset of the code is available in Appendix B. We did not find any errors in this implementation. However, we did introduce various errors, and the assertions failed as expected.

7. Related and Future Work

In this section we discuss related work and future work.

7.1 Related Work

There has been a great deal of research on static shape analysis using abstract interpretation, data flow analysis, type systems, model checking and various different kinds of logics. Efforts in this direction are complementary to the dynamic shape analysis we propose in this paper. As mentioned in the introduction, the inspiration for using a substructural logic to describe heap shape comes from the work of O'Hearn, Reynolds, Yang and others on separation logic [22] — a technique for static verification of pointer programs. Our choice of linear logic is mostly out of practical concerns; since LolliMon is available for us to use and modify.

Tools for dynamic verification of heap properties include Purify [5] and SWAT [6]. Purify traps every memory access call in a program and detects generic problems such as dangling pointer access. SWAT uses a profiling infrastructure to monitor memory access operations and detect memory leaks. 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.

7.2 Future Work

In the future, we plan to continue working on our assertion language in order to make it easier 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.

Finally, we are currently engaged in research on static verification of program properties using linear logic [1, 10, 11]. In the long term, we hope to use the same language of heap-shape assertions both as dynamic contracts and as annotations for static verification.

8. Conclusions

In this paper, we show how to use linear logic as a language of *contracts* that document and enforce heap-shape properties in imperative programs. Unlike the ad hoc, unstructured heap-shape contracts one might write in native code, linear logical contracts are

clear, concise and have a well-defined semantics. We have implemented a rather substantial C-like programming language that includes linear logical heap-shape contracts. The contracts are evaluated at runtime using a modified version of the LolliMon logic programming engine. Using our implementation, we have experimented with contracts for a variety of data structures including linked lists, doubly-linked lists, circular lists, trees and DAGs. Several of these examples are explained in this paper. Finally, we have developed an indexed heap model for linear logic with inductive definitions and proven that linear logic's proof theory, and therefore our logic engine, is sound with respect to this model.

References

[1] 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.

[2] 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.

[3] 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.

[4] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.

[5] 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.

[6] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.

[7] 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.

[8] R. C. Holt and J. R. Cordy. The Turing programming language. *Commun. ACM*, 31(12):1410–1423, 1988.

[9] 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.

[10] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, 2005.

[11] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *European Symposium on Programming Languages*, Apr. 2006.

[12] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *PPDP '05*, pages 35–46, New York, NY, USA, 2005. ACM Press.

[13] P. López and J. Polakow. Implementing efficient resource management for linear logic programming. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 528–543, 2004.

[14] D. C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying ADA Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[15] B. Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22(2):85–94, 1987.

[16] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[17] G. Morrisett, A. Ahmed, and M. Fluet. L^3: A linear language with locations. In *Seventh International Conference on Typed Lambda Calculi and Applications*, Apr. 2005.

[18] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[19] 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.

[20] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.

[21] Programming with assertions. http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html.

[22] 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, 2002.

A. Sequent Calculus for Intuitionistic Linear Logic

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

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

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

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

$$\frac{\Gamma; \Delta, F_1 \longrightarrow F_2}{\Gamma; \Delta \longrightarrow F_1 \multimap F_2}\ \multimap R$$

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

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

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

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

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

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

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

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

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

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

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

B. Redblacktree.minic

Below we include partial code for the MiniC implementation of red-black trees [2]. 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.

[2] based on code from http://oopweb.com/Algorithms/Documents/Sman/Volume/s_rbt.txt

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

[[
checkData: int -> int -> int -> o.
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 leaf nodes are sentinels */
node_tp *sentinel;

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


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

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 ommitted...
  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 node -- includes invariant check */
  status = delete(3);
}
```