Linear Logic, Heap-shape Patterns and Imperative Programming

Extended Abstract

Limin Jia
Princeton University
ljia@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

In this paper, we propose a new programming paradigm designed to simplify the process of safely creating, manipulating and disposing of complex mutable data structures. In our system, programmers construct data structures by specifying the shapes they want at a high level of abstraction, using linear logical formulas rather than low-level pointer operations. Likewise, programmers deconstruct data structures using a new form of pattern matching, where the patterns are again drawn from the syntax of linear logic. In order to ensure that algorithms for construction, inspection and deconstruction of heap values are well-defined and safe, we analyze the programmer's linear logical specifications using a mode analysis inspired by similar analysis used in logic programming languages. This mode analysis is incorporated into a broader type system that ensures the memory safety of the overall programming language. We have implemented the language and explored using it to manipulate a variety of data structures including lists, trees, and an adjacency list representation of graphs.

1. Introduction

One of the most important and enduring problems in programming languages research involves verification of programs that construct, manipulate and dispose of complex heap-allocated data structures. Any solution to this difficult problem can be used to guarantee memory safety properties and as a foundation for the verification of higher-level program properties.

Over the last several years, great progress has been made on this problem by using substructural logics to specify the shape of heap-allocated data structures [18, 12, 16]. The key insight is that these logics can capture aliasing properties in a substantially more concise notation than is possible in conventional logics. This new notation makes proofs more compact, and easier to read, write and understand. One notable example is O'Hearn, Reynolds, Yang and others' work on separation logic [18]. These authors specify heap-shape invariants using a variant of the logic of bunched implications (BI) [17]. They then include the BI specifications in a Hoare logic to verify the correctness of low-level pointer programs.

O'Hearn's process is a highly effective way of verifying existing pointer programs. However, if one needs to construct new software with complex data structures, there are opportunities for simplifying and improving the combined programming and verification process. In particular, writing low-level pointer programs remains tricky in O'Hearn's setting. Verifying the data structures one has created using separation logic provides strong safety and correctness guarantees at the end of the process, but it does not simplify, speed up, or prevent initial mistakes in the programming task.

In this paper, we propose a new programming paradigm designed to simplify the combined process of constructing data structures and verifying that they meet complex shape specifications. We do so by throwing away the low-level pointer manipulation statements, leaving only the high-level specifications of data structure shape. So rather than supporting a two-step program-then-verify paradigm, we support a one-step correct-by-construction process.

More specifically, programmers create data structures by specifying the shapes they want in linear logic (a very close relative of O'Hearn's separation logic).These linear logical formulas are interpreted as algorithms that allocate and initialize data structures desired by the programmer. To use data, programmers write pattern-matching statements, somewhat reminiscent of ML-style case statements, but where the patterns are again formulas in linear logic. Another algorithm takes care of matching the linear logical formula against the available storage. To update data structures, programmers simply specify the structure and contents of the new shapes they desire. The run-time system reuses heap space in a predictable fashion. Finally, a "free" command allows programmers to deallocate data structures as they would in an imperative language like C. In order to ensure that algorithms for construction, inspection and deconstruction of heap values are well-defined and memory-safe, we analyze the programmer's linear logical specifications using a mode analysis inspired by similar analysis used in logic programs. This mode analysis is incorporated into a broader type system that ensures the safety of the overall programming language.

In summary, this paper makes the following contributions:

1. It develops novel algorithms used at run time to interpret linear logical formulas as programs to allocate and manipulate complex data structures.
2. It develops a new mode analysis for linear logical formulas that helps guarantee these algorithms are safe – they do not dereference dangling pointers.
3. It shows how to incorporate these run-time algorithms and static analysis into a safe imperative programming language.
4. All of the examples in this paper and more, have been implemented and verified by the system.

Overall, the result is a new programming paradigm in which linear logical specifications, rather than low-level pointer operations, drive safe construction and manipulation of sophisticated heap-allocated data structures.

The rest of the paper is organized as follows: In Section 2 we give an informal overview of our system, show how to define the shape invariants of recursive data structures using linear logic, and explain the basic language constructs that construct and deconstruct heap shapes. Next, in Section 3, we delve into the details of the algorithmic interpretations of the logical definitions for heap shapes and the mode analysis for preventing illegal memory operations. In Section 4, we introduce the formal syntax, semantics, and the type system for the overall language. In Section 5, we illustrate the extent of the language's expressive power by explaining how to define adjacency list representation of graphs. Finally, we discuss

related work. Space constraints only allow us to sketch the main ideas involved in defining the formal semantics.

2. System Overview

The main idea behind our system is to give programmers the power of using linear logic to define and manipulate recursive data structures. In this section, first, we introduce the program heap and the basic describing formulas of the heap. Then, we explain how to define recursive data structures using linear logic. Next, we show how to use the logical definitions to manipulate data structures in our language. Finally, we informally explain the invariants that keep our system memory safe.

2.1 The Heap

The program heap is a finite partial map from locations to tuples of integers. Locations are themselves integers, and 0 is the special `NULL` pointer. Every tuple consists of a header word followed by some data. The header word stores the size (number of elements) of the rest of the tuple. We often use the word *heaplet* to refer to a fragment of a larger heap. Two heaplets are *disjoint* if their domains have no locations in common.

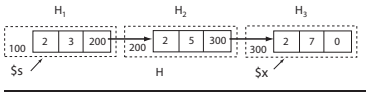

Figure 1. Memory containing a linked list.

As a simple example, consider the heap `H` in Figure 1, which we will refer to throughout this section. It is composed of three disjoint heaplets: $\mathtt{H}_1$, $\mathtt{H}_2$, and $\mathtt{H}_3$. Heap $\mathtt{H}_1$ maps location 100 to tuple $(2, 3, 200)$, where the integer 2 in the first field of the tuple indicates the size of the rest of the tuple.

We use $dom(\mathtt{H})$ to denote the set of locations in `H`, and $\bar{dom}(\mathtt{H})$ to denote the set of starting locations of each tuple in `H`. We write $\mathtt{H}(l)$ to represent the value stored in location l, and $\bar{\mathtt{H}}(l)$ to represent to tuple stored at location l. For example, for `H` in Figure 1, $\bar{dom}(\mathtt{H}) = \{100, 200, 300\}$, and $\bar{\mathtt{H}}_1(100) = (3, 200)$. We use $\mathtt{H}_1 \uplus \mathtt{H}_2$ to denote the union of two disjoint heaplets $\mathtt{H}_1$ and $\mathtt{H}_2$. It is undefined if $\mathtt{H}_1$ and $\mathtt{H}_2$ are not disjoint.

2.2 Basic Descriptions

Programmers describe heaps and heaplets using a collection of domain-specific predicates together with formulas drawn from linear logic. In order to describe individual tuples, programmers use the predicate `node x T`, where `x` is the starting address and `T` is the tuple contents. For example, `(node 100 (3, 200))` describes heaplet $\mathtt{H}_1$. To describe larger structures, programmers use a multiplicative conjunction (identical to separation logic's multiplicative conjunction "$*$") written "," . Formula $(\mathtt{F}_1, \mathtt{F}_2)$ describes a heap composed of two disjoint heaplets $\mathtt{H}_1$ and $\mathtt{H}_2$ such that $\mathtt{H}_1$ can be described by $\mathtt{F}_1$, and $\mathtt{H}_2$ can be described by $\mathtt{F}_2$. For example, heap `H` can be described by `(node 100 (3, 200), node 200 (5, 300), node 300 (7, 0))`. Programmers use additive disjunction, written ";", to combine multiple possible descriptions. Formula $(\mathtt{F}_1; \mathtt{F}_2)$ describes a heap that can be described by either $\mathtt{F}_1$ or $\mathtt{F}_2$. Programmers may also describe finer-grained properties of their data structures using integer inequality and set constraints.

In addition to using these primitive descriptions, programmers can create new definitions to describe shapes. In the next section, for instance, we will show how to define lists, queues and trees.

2.3 Logical Shape Signatures

A logical *shape signature* is a set of definitions that collectively defines algorithms for run-time manipulation of complex data structures and proof rules for compile-time checking. Each shape signature contains three basic elements: *inductive definitions*, which define shape structure and run-time algorithms; *axioms*, which give relations between shapes and are used during compile-time type checking; and *type and mode declarations*, which constrain the kinds of inductive definitions allowed so as to ensure the corresponding run-time algorithms are both memory-safe and well-defined. In the following subsections we explain each part of the signature in turn.

Inductive Definitions. In order to define the basic shapes of data structures, we borrow technology and notation from the field of linear logic programming [10, 14]. The inductive definitions are written down as a series of clauses that mimic a linear logic program. Each clause is composed of a head (a predicate such as `list X`), followed by the inverted linear implication "o-," followed by the body of the clause (a basic description that references the head or other newly defined predicates). Free variables appearing in the head may be viewed as universal parameters to the definition; free variables appearing in the body are existentially quantified. Definitions are terminated with a period.

As an example, consider defining a null-terminated non-circular singly linked list starting from address `X`. The principal clause for `list X` is given by the following statement:

```
list X o- (X = 0); (node X (D, Y), list Y).
```

The body is the additive disjunction of two cases. The first case says that 0 is a list pointer; the second one says that X is a list pointer if it points to a pair of values D and Y such that Y is a list pointer. Notice that the head and the tail of the list are separated by ",", and therefore, they are two disjoint pieces of the heap. This constraint guarantees the list will be non-circular.

A closely related definition, `listseg X Y`, can be used both to reason about lists and to help us define another data structure, the queue. The definition for `listseg X Y` describes a non-circular singly linked list segment starting from location X and ending at Y.

```
listseg X Y o- (X=Y); (not (X=Y), node X (D, Z), listseg Z Y).
```

The base case states that `listseg X X` is always true; the second case states that if `X` points to a pair of values `D` and `Z` such that between `Z` and `Y` is a list segment, then between `X` and `Y` is also a list segment. The inequality of `X` and `Y` together with the disjointness of the head and tail of the list segment guarantees non-circularity.

The next example uses the `listseg` predicate to define a queue.

```
queue X Y o- ((X = 0), (Y = 0)); (listseg X Y, node Y (D,0)).
```

The predicate `queue X Y` describes a queue whose head is `X` and tail is `Y`. In the clause above, the first case describes the situation when the queue is empty and both the head and tail pointers are 0. The second case describes the situation in which there is at least one element in the queue (pointed to by `Y`). Between the head and the tail of the queue is a `listseg`. For example, the heap `H` in Figure 1 can be viewed as a queue whose head pointer is $s (*i.e.*, location 100), and tail pointer is $x (*i.e.*, location 300).

We can define tree-shaped data similarly. As an example, consider the following binary tree definition.

```
btree X o- (X = 0); (node X (D, L, R), btree L, btree R).
```

Axioms. Each shape signature can contain many inductive definitions. For instance, the `listshape` signature we will be using as a running example through this paper will contain both the definitions of `list` and `listseg`. In order to allow the system to reason about the relationships between these various definitions, the programmer must write down additional clauses, which we call axioms. For example, the following axiom relates `list` to `listseg`.

```
list Y o- listseg Y Z, list Z.
```

Without this axiom, the type system cannot prove that one complete shape, such as `(listseg x y, list y)`, is related to another (`list x`). While the syntax of axioms is very similar to that for inductive

definitions, the axioms are used for compile time reasoning only, not generation of algorithms for construction and deconstruction of data structures. Hence, the form that axioms can take is slightly more liberal than that of inductive definitions.

Type and Mode Declarations. All predicates are given both types and modes. The purpose of the types is to constrain the sorts of data (*e.g.*, either pointers or integers) that may appear in particular fields of a data structure. The purpose of the modes is to ensure that the heap-shape pattern-matching algorithm is safe and efficient.

To understand the purpose of mode declarations and mode analysis, consider the problem of matching the predicate `node X (77,34)` against the contents of some heap `H`. Logically, the goal of the matching algorithm is to find an address `l` such that `H(l) = (77,34)`. However, without any additional information, it would seem the only possible algorithm would involve examining the contents of every address in the entire heap `H` until one that satisfies the constraint is found. But of course, in general, attempting to use such an algorithm in practice is hopelessly inefficient.

On the other hand, suppose we are given a specific address `l'` and we would like to match the formula (`node l' (D,X), node X (77,34)`) against some heap `H`. We can simply look up `l'` in `H` to determine values for `D` and `X`. The value of `X` is subsequently used to determine whether `H(X)=(77,34)`. We also need to ensure the value of `X` is not equal to l' (otherwise the linearity constraint that `l'` and `X` point to disjoint heaplets would be violated).

When a value such as `l'` or `X` is known, it is referred to as *ground*. *Mode declarations* specify, among other things, expectations concerning which variables are ground in which positions. Finally, *mode analysis* is a syntactic analysis, much like type checking, that can determine whether the mode declarations are correct.

In our system, the modes for specifying groundness conditions are the standard ones found in many logic programming languages. In particular, the input mode (`+`) specifies that a term in that position must be ground before evaluation of the predicate. The output mode (`-`) specifies the term in that position must be ground term after the predicate is evaluated. The last mode (`*`) indicates we do not care about this position. Now, to guarantee it is possible to evaluate the predicate `node X (...)` in constant time, we give the first position (`X`) the input mode (`+`). Once the first argument of the `node` predicate has been constrained to have input mode, other definitions that use it are constrained in turn. For example, the first arguments of `list X` and `queue X Y` must also be inputs.

Ensuring pointers are ground before lookup provides a guarantee that lookup will occur in constant time. However, it does not guarantee that the pointer in question points to a valid heap object. For example, when the matching algorithm attempts to match predicate `node l (...)` against a heap `H`, `l` is not necessarily a valid address in H. A second component of our mode analysis characterizes pointers as either `sf` (definitely not dangling) or `unsf` (possibly dangling) or `unsf sf` (possibly dangling before evaluation of the predicate, but definitely not dangling if the predicate is successfully evaluated), and thereby helps guarantee the matching algorithm does not go wrong. The last safety mode (`unsf sf`) is used when the evaluation of the predicate has allowed us to learn that a particular pointer is safe.

The complete mode for arguments of pointer type is a pair $(\mathtt{g}, \mathtt{s})$, where `g` describes the argument's groundness property, and `s` describe its safety property. Integers are not dereferenced and hence their modes consist only of the groundness condition `g`.

As an example, the combined type and mode declaration for lists follows. It states that the list predicate must be supplied with a single ground, non-dangling pointer argument.

```
list : (+,sf) ptr(node) -> o.
```

```
listshape {
 struct node : (+,sf) ptr(node) -> (- int, (-,sf) ptr(node))-> o.
 listshape : (+,sf) ptr(node) -> o.
 list      : (+,sf) ptr(node) -> o.
 listseg   : (+,sf) ptr(node) -> (+,unsf sf) ptr(node) -> o.

 listshape X o- list X.
 list X o- (X = 0); (node X (D,Y), list Y).
 listseg X Y o- (X = Y); not(X = Z), node X (D,Y), listseg Y Z.
 with
 list Y o- listseg Y Z, list Z.}
```

Figure 2. Singly Linked List Shape Signature

Putting the Declarations Together. Figure 2 is the full shape signature for `listshape`. The first definition gives the structure of the tuples that are to be allocated in memory (the "struct" keyword is used to indicate that `node` predicate will be realized as a concrete piece of data). We also call these predicates "struct" predicates. This definition and the next three define the modes for the inductive definitions. The next three are inductive definitions used to create data structures. The last definition (separated from the others using the keyword "with") gives an axiom for relating lists and listsegs.

2.4 The Programming Language

In this section we explain how to incorporate the logical definitions of data structures into a safe, imperative programming language.

2.4.1 Basic Language Structure

A program is composed of a collection of shape signatures and function definitions. Program execution begins with the distinguished "main" function. Within each function, programmers declare, initialize, use and update local imperative variables (also referred to as "stack variables"). Each such variable is given a basic type, which may be an integer type (**int**), a shape type, or a pointer type. The shape types, such as `listshape`, are named by the shape signatures. The pointer types, such as $\mathbf{ptr}(\mathtt{node})$, specify the specific kind of tuple a pointer points to. In order to distinguish the logical names `X`, `Y`, `Z`, etc. introduced via logical pattern matching, from the imperative variables, we precede the names of imperative variables with a $ sign. We use $\$s$ to range over shape variables and $\$x$ to range over integer or pointer variables.

2.4.2 Operations on Shapes

As discussed earlier, formulas describing the heap serve both to help programmers create new shapes and to deconstruct, or disassemble, existing shapes.

Creating Shapes. Creating data structures with certain shapes is done using the shape assignment statement as shown below.

```
$s:listshape := {a1, a2, a3}[root a1, node a1 (3, a2),
                             node a2 (5, a3), node a3 (7, 0)]
```

The right-hand side of a shape assignment describes the shape to be created and the left-hand side specifies the imperative shape variable to be assigned. In this case, we will assume the shape variable $\$s$ has `listshape` type (see Figure 2).

Variables `a1`, `a2`, and `a3` in the braces indicate the new tuples to be allocated on the heap to make the formula in braces a valid `listshape`. The size of each tuple is determined by examining the type declaration of the `node` predicate in `listshape` signature. Each variable is subsequently bound to the address of the corresponding tuple.

Once space has been allocated, the integer data fields are initialized with the values appearing in the shape description. Finally, the location specified by the `root` predicate is stored into the shape variable $\$s$. This special `root` predicate indicates the starting address of this shape and must always appear in all shape descriptions.

Deconstructing Shapes and Reusing Deconstructed Shapes. To deconstruct a shape, we use a pattern-matching notation. For example, to deconstruct the list contained in the imperative variable $\$s$, we might use the following pattern:

```
$s:[root r, node r (d, next), list next]
```

This pattern, when matched against the heaplet reachable from $\$s$, may succeed and bind `r`, `next` and `d` to values, or it may fail. If it succeeds, `r` will be bound to the pointer stored in $\$s$, `d` will be bound to integer data from the first cell of the list, and `next` will be bound to a pointer to the next element in the list.

Pattern matching does not deallocate data. Consequently, it is somewhat similar to the unrolling of a recursive ML-style datatype, during which we change our view of the heap from an abstract shape (*e.g.*, a `listshape`), to a more descriptive one (*e.g.*, a pointer to a pair of values `d` and `next`, where `next` points in turn to a list). More formally, the unrolling corresponds to revealing that the heaplet in question satisfies the following elaborate formula:

```
∃r.∃d.∃next.(root r, node r (d, next), list next)
```

Pattern matching occurs in the context of branching statements in our language. Here is an example of an if statement.

```
if $s:[root r, node r (d, next), list next]
then {free r; $s := [root next, list next]}
else print ''list is empty''
```

In evaluating the if statement, first we evaluate shape pattern. If the pattern match succeeds, then a substitution for the bound variables is returned, the substitution is applied on the true branch, and evaluation of the true branch continues. If the pattern matching fails, then the false branch is taken. The variables in the shape pattern are not in scope in the false branch. Suppose $\$s$ points to the first tuple in the heap `H` displayed in Figure 1. When the shape pattern is evaluated, `r` will be bound to 100, `d` will be bound to 3, and `next` will be bound to 200. The execution will proceed with evaluation of the true branch, where we free the first tuple of the list, then reconstruct a list using the rest of the old list. The predicate `root next` specifies the root of this new shape. Operationally, the run-time value of `next`, 200, is stored in the variable $\$s$.

2.5 An Example Program

```
1    listshape delete(listshape $s, int $k){
2      ptr(node) $pre := 0;
3      ptr(node) $p := 0;
4      if $s?[root x, list x]
5      then {$pre := x; $p := x}
6      else skip;
7      while ($s?[root x, listseg x $p,
8                node $p (key, next), list next,
9                not($k = key) ])
10     do {$pre := $p; $p := next};
11     switch $s of
12        :[root x, y = $p, node x (d, nxt), list nxt, d = $k ]
13        -> {free x; $s:listshape := [root nxt, list nxt]}
14       |:[root x, y = $pre, z = $p, listseg x y,
15          node y (dy, z), node z (dz, next), list next, dz = $k]
16        ->{free z;
17           $s:listshape :=
18             [root x, listseg x y, node y (dy, next), list next]
19       |_ -> skip;
20     return $s; }
```

Figure 3. list_delete

As an example of our language in action, consider the function delete, which removes an integer from a list. The first argument of `delete`, `$s`, has `listshape` type and holds the starting address of the list. The second argument, `$k`, is the integer to be deleted. The algorithm uses pointer `$p` to traverse the list until it reaches the end of the list or the data under `$p` is equal to the key `$k` to be deleted. A second pointer `$pre` points to the parent of `$p`. The if statement between line 4 and 6 initializes both `$p` and `$pre` to point to the head of the list. The while loop between lines 7 and 10 walks down the list maintaining the invariant expressed in the while condition in each iteration of the loop. This invariant states that (1) the initial part of the list is a `listseg` ending with pointer `$p`, (2) `$p` points to a node that contains a key and a next pointer, (3) the next pointer itself points to a list and (4) `key` is not the key. When either condition (2) or (4) is falsified, control breaks out of the loop (conditions (1) and (3) cannot be falsified). The switch statement between line 11 and 19 deletes the node from the list (if a node has been found). The first branch in the switch statement covers the case when the node to be deleted is the head of the list; the second branch covers the case when the node to be deleted is pointed to by `$p`; the last (default) branch covers the case when `$k` is not present in the list.

2.6 What Could Go Wrong

Adopting a low-level view of the heap and using linear logic to describe recursive data structure gives our language tremendous expressive power. However, the expressiveness calls for an equally powerful type system to deliver memory-safety guarantees. We have already mentioned some of the elements of this type system, including mode checking for logical declarations, and the use of inductive definitions and axioms to prove data structures have the appropriate shapes. In this section, we summarize several key properties of the programming language's overall type system, what could go wrong if these properties are missing, and what mechanisms we use to provide the appropriate guarantees.

Safety of Deallocation. Uncontrolled deallocation can lead to double freeing and dereferencing dangling pointers. We must make sure programmers do not use the deallocation command too soon or too often. To provide this guarantee, our type system keeps track of and describes (via linear logical formulas) the accessible heap, in much the same way as O'Hearn's separation logic or its closely related type systems [24, 25, 4, 1, 26]. In all cases, linearity constraints separate the description of one data structure from another to make sure that the effect of deconstruction and reconstruction of shapes is accurately represented.

Safety of Dereferencing Pointers. Pointers are dereferenced when a shape pattern-matching statement is evaluated. The algorithm could potentially dereference dangling pointers by querying ill-formed shape formulas. Consider the following pattern:

```
$s: [root r, node 12 (d, 0), node r (dr, 12)]
```

Here there is no reason to believe "12" is a valid pointer. Predicate mode and type checking prevents programmers from writing such ill-formed statements.

Termination for Heap Shape Pattern Matching As we saw in the examples, the operational semantics invokes the pattern-matching procedure to check if the current program heap satisfies certain shape formulas. It is crucial to have an efficient and tractable algorithm for the pattern-matching procedure. In our system, this pattern-matching procedure is generated from the inductive definitions in the logic signature, and uses a bottom-up, depth-first algorithm. However, if the programmer defines a predicate `Q` as `Q X o- Q X`, then the decision procedure will never terminate. To guarantee termination, we place a well-formedness restriction on the inductive definitions that ensures a linear resource is consumed before the decision procedure calls itself recursively. Our restriction rules out the bad definition of `Q` and others like it.

Term	tm	::=	$x \mid n \mid -\mathtt{tm} \mid \mathtt{tm} + \mathtt{tm}$
Arith Pred	Pa	::=	$\mathtt{tm} = \mathtt{tm} \mid \mathtt{tm} > \mathtt{tm}$
Arith Formula	A	::=	$\mathtt{Pa} \mid \mathtt{not\ Pa}$
User-Defined Pred	Pu	::=	$\mathbf{P}\ \overline{\mathtt{tm}}$
Literals	L	::=	$\mathtt{A} \mid \mathtt{Ps\ tm}\ (\overline{\mathtt{tm}}) \mid \mathtt{Pu}$
Formulas	F	::=	$\mathtt{emp} \mid \mathtt{L}, \mathtt{F}$
Inductive Def/Axiom	I/Ax	::=	$(\mathtt{Pu}\ \mathtt{o}\!-\ \overline{\mathtt{F}})$
Groundness	g	::=	$+ \mid - \mid *$
Safety Qualifier	s	::=	$\mathtt{sf} \mid \mathtt{unsf} \mid \mathtt{unsf\ sf}$
Mode	m	::=	$\mathtt{g} \mid (\mathtt{g}, \mathtt{s})$
Arg Type	argtp	::=	$\mathtt{g}\ \mathbf{int} \mid (\mathtt{g}, \mathtt{s})\ \mathbf{ptr}(\mathbf{P})$
Pred Type	pt	::=	$\mathtt{o} \mid \mathtt{argtp} \rightarrow \mathtt{pt} \mid (\overline{\mathtt{argtp}}) \rightarrow \mathtt{pt}$
Pred Type Decl	pdecl	::=	$\mathbf{P}\ :\ \mathtt{pt} \mid \mathtt{struct\ Ps}\ :\ \mathtt{pt}$
Shape Signature	SS	::=	$\mathbf{P}\{\ \overline{\mathtt{pdecl}}.\ (\mathbf{P}\ x\ \mathtt{o}\!-\ \mathtt{F}).\ \overline{\mathtt{I}}.\ \overline{\mathtt{Ax}}\}$
Pred Typing Ctx	Ξ	::=	$\cdot \mid \Xi, \mathbf{P}{:}\mathtt{pt}$
SS Context	Λ	::=	$\cdot \mid \Lambda, \mathbf{P}{:}\Xi$
Logical Rules Ctx	Υ	::=	$\cdot \mid \Upsilon, \mathtt{I} \mid \Upsilon, \mathtt{Ax}$

Figure 4. Syntax of Logical Constructs

2.7 Three Additional Caveats

For the system as a whole to function properly, programmers are required to check the following three properties themselves.

Closed Shapes. A closed shape is a shape from which no dangling pointers are reachable. For example, lists, queues and trees are all closed shapes. On the other hand, the `listseg` definition given earlier is not closed—if one traverses a heaplet described by a `listseg`, the traversal may end at a dangling pointer. Shape signatures may contain inductive definitions like `listseg`, but the top-level shape they define must be closed. If it is, then all data structures assigned to shape variables $\$s$ will also be closed and all pattern-matching operations will operate over closed shapes. This additional invariant is required to ensure shape pattern matching does not dereference dangling pointers.

Soundness of Axioms. For our proof system to be sound with regard to the semantics, the programmer-defined axioms must be sound with respect to the semantics generated by the inductive definitions. As in separation logic, checking properties of different data structures requires different axioms and programmers must satisfy themselves of the soundness of the axioms they write down and use. We have proven the soundness of all the axioms that appear in this paper (and axioms relating to other shapes not in the paper).

Uniqueness of Shape Matching. Given any program heap and a shape predicate with a known root location, at most one heaplet should match the predicate. For example, given the heap `H` in Figure 1, predicate `list 200` describes exactly the portion of `H` that is reachable from location 200, ending in `NULL` ($\mathtt{H}_2, \mathtt{H}_3$). Without this property, the operational semantics would be nondeterministic. Programmers must verify this property themselves by hand. Once again, it holds for all shapes described in this paper.

These requirements are not surprising. For instance, separation logic's specialized axioms concerning lists and trees must be verified separately as well. The requirement concerning uniqueness of shapes are very similar to the *precise predicates* used in the work on separation and information hiding [19]. These requirements could well be the common invariants substructural logical systems should have when used in program verification.

3. Logical Shape Signatures: Formal Semantics

3.1 Syntax

The syntactic constructs of our logic are listed in Figure 4. Throughout the paper we use the overbar notation $\overline{x}$ to denote a vector of objects x. We use `tm` to range over terms. Arithmetic formulas include arithmetic predicates, which are the equality and partial order of terms, and their negations. We use `Ps` to range over all the "struct" predicates such as `node`, $\mathbf{P}$ to range over user-defined predicate names such as `list`, and `Pu` to range over fully applied user-defined predicates. A literal `L` can be either an arithmetic formula or a state predicate or a user-defined predicate. We use `F` to range over formulas which are either `emp` (the empty heap) or the conjunction of a literal and another formula.

The head of a clause is a user-defined predicate and the body is a formula. For the ease of type checking, we gather the bodies of the same predicate into one definition `I`. The notation $\overline{\mathtt{F}}$ means the additive disjunction of all the $\mathtt{F}_i$ in $\overline{\mathtt{F}}$. Axioms are also clauses.

A simple argument type is a mode followed by the type of the argument. Argument types for predicate can either be simple argument types or a tuple of the simple argument types. A fully applied predicate has type `o` (a standard way of writing the type of logical formulas).

Context Ξ contains all the predicate type declarations in one shape signature. Context Λ maps each shape name to a context Ξ. Lastly, context Υ contains all the inductive definitions and axioms defined in the program.

3.2 Store Semantics

We use $\mathtt{H} \vDash^{\Upsilon} \mathtt{F}$ to mean that heap `H` can be described by formula `F`, under the inductive definitions in Υ. Since Υ is fixed throughout, we omit it from the judgments. We use the notation $\Upsilon(\mathbf{P})$ to denote the inductive definitions for $\mathbf{P}$. We present the formal definition of the store semantics of our logic in Figure 5. An arithmetic formula `A` is valid if the heap is empty and `A` is valid. Heap `H` satisfies $\mathtt{Ps\ v\ (v_1 \ldots v_n)}$, if the domain of `H` contains exactly the $n+1$ consecutive locations starting from `v`, and the first location of `H` stores the size of the tuple, and values $\mathtt{v_1}$ through $\mathtt{v_n}$ match the contents of the heap. To give a properly inductive semantics to user-defined predicates, we index the predicates with a natural number n. The semantics of an unrolling of an inductive definition depends on predicates with a strictly smaller index.

3.3 Logical Deduction

Type checking requires reasoning in linear logic with the inductive definitions and axioms the user has defined. A formal logical deduction in our system has the form: $\Gamma; \Delta \Longrightarrow \mathtt{F}$. Γ is the unrestricted context (hypotheses may be used any number of times) and Δ is the linear context (hypothesis can be used exactly once). Initially, Γ will be populated by the inductive definitions and axioms from the shape signatures.

We have proved that our logical deduction system is sound with respect to the semantics modulo the soundness of axioms.

Lemma 1 (Soundness of Logical Deduction)
Assume the user-defined axioms (Υ_A) are sound with respect to the store semantics defined by (Υ_I), and $\Upsilon = \Upsilon_I, \Upsilon_A$ and $\Upsilon; \Delta \Longrightarrow \mathtt{F}$, and σ is a

- $\mathtt{H} \vDash \mathtt{emp}$ iff $\mathtt{H} = \emptyset$.
- $\mathtt{H} \vDash \mathtt{A}$ iff $\mathtt{H} = \emptyset$, and `A` is a valid arithmetic formula.
- $\mathtt{H} \vDash \mathtt{Ps}\ v\ (v_1, \cdots, v_n)$ iff $v \neq 0$, $dom(\mathtt{H}) = \{v, v+1, \cdots v+n\}$, $\mathtt{H}(v) = n$, and for all $i \in [1, n]$, $\mathtt{H}(v+i) = v_i$.
- $\mathtt{H} \vDash \mathtt{F}_1, \mathtt{F}_2$ iff $\mathtt{H} = \mathtt{H}_1 \uplus \mathtt{H}_2$, such that $\mathtt{H}_1 \vDash \mathtt{F}_1$, and $\mathtt{H}_2 \vDash \mathtt{F}_2$.
- $\mathtt{H} \vDash \mathtt{F}_1; \mathtt{F}_2$ iff $\mathtt{H} \vDash \mathtt{F}_1$ or $\mathtt{H} \vDash \mathtt{F}_2$.
- $\mathtt{H} \vDash \mathbf{P}\ \overline{v}$ iff $\exists n$ st. $\mathtt{H} \vDash \mathbf{P}^n\ \overline{v}$
- $\mathtt{H} \vDash \mathbf{P}^0\ \overline{v}$ iff $\Upsilon(\mathbf{P}) = \mathbf{P}\ \overline{x}$ o- $\overline{\mathtt{F}}$, and $\exists i$ st. $\mathtt{H} \vDash \mathtt{F}_i\,[\,\overline{v}\,/\,\overline{x}\,]$, and $\nexists \mathbf{P} \in dom(\mathtt{F}_i)$. where the free variables in $\mathtt{F}_i$ are considered existentially quantified.
- $\mathtt{H} \vDash \mathbf{P}^n\ \overline{v}$ iff $\Upsilon(\mathbf{P}) = \mathbf{P}\ \overline{x}$ o- $\overline{\mathtt{F}}$, and $\exists i$ st. $\mathtt{H} \vDash \mathtt{F}_i\,[\,\overline{v}\,/\,\overline{x}\,]$ and $n-1$ is the maximum of the index number in $\mathtt{F}_i$.

Figure 5. Selected Rules of the Semantics for Formulas

$\boxed{\Pi \vdash \mathtt{F} : \Pi'}$

$$\frac{\mathtt{mode}(\mathbf{P}) = (+, \mathtt{sf}) \rightarrow \mathtt{o} \quad x{:}\mathtt{sf} \in \Pi}{\Pi \vdash \mathbf{P}\ x : \Pi}\ \textit{mode-1} \qquad \frac{\mathtt{mode}(\mathbf{P}) = (+, \mathtt{unsf}) \rightarrow \mathtt{o} \quad x{:}\mathtt{s} \in \Pi}{\Pi \vdash \mathbf{P}\ x : \Pi}\ \textit{mode-3}$$

$$\frac{\mathtt{mode}(\mathbf{P}) = (-, \mathtt{sf}) \rightarrow \mathtt{o}}{\Pi \vdash \mathbf{P}\ x : \Pi \bar{\cup} (x{:}\mathtt{sf})}\ \textit{mode-2} \qquad \frac{\mathtt{mode}(\mathbf{P}) = (-, \mathtt{unsf}) \rightarrow \mathtt{o}}{\Pi \vdash \mathbf{P}\ x : \Pi \bar{\cup} (x{:}\mathtt{unsf})}\ \textit{mode-4}$$

$$\frac{\Pi \vdash \mathtt{L} : \Pi' \quad \Pi' \vdash \mathtt{F} : \Pi''}{\Pi \vdash \mathtt{L}, \mathtt{F} : \Pi''}\ \textit{mode-5}$$

$\boxed{\vdash \mathbf{P}\ x\ \mathtt{o}{-}\ \mathsf{OK}}$

$$\frac{\mathtt{mode}(\mathbf{P}) = (+, \mathtt{sf}) \rightarrow \mathtt{o} \quad x{:}\mathtt{sf} \vdash \mathtt{F} : (x{:}\mathtt{sf}), \Pi}{\vdash \mathbf{P}\ x\ \mathtt{o}\text{-}\mathtt{F} : \mathsf{OK}}\ \textit{mode-6} \qquad \frac{\mathtt{mode}(\mathbf{P}) = (+, \mathtt{unsf}) \rightarrow \mathtt{o} \quad x{:}\mathtt{unsf} \vdash \mathtt{F} : (x{:}\mathtt{s}), \Pi}{\vdash \mathbf{P}\ x\ \mathtt{o}\text{-}\mathtt{F} : \mathsf{OK}}\ \textit{mode-7}$$

$$\frac{\mathtt{mode}(\mathbf{P}) = (+, \mathtt{unsf\ sf}) \rightarrow \mathtt{o} \quad x{:}\mathtt{unsf} \vdash \mathtt{F} : (x{:}\mathtt{sf}), \Pi}{\vdash \mathbf{P}\ x\ \mathtt{o}\text{-}\mathtt{F} : \mathsf{OK}}\ \textit{mode-8}$$

$$\frac{\mathtt{mode}(\mathbf{P}) = (-, \mathtt{sf}) \rightarrow \mathtt{o} \quad \vdash \mathtt{F} : (x{:}\mathtt{sf}), \Pi}{\vdash \mathbf{P}\ x\ \mathtt{o}\text{-}\mathtt{F} : \mathsf{OK}}\ \textit{mode-9} \qquad \frac{\mathtt{mode}(\mathbf{P}) = (-, \mathtt{unsf}) \rightarrow \mathtt{o} \quad \vdash \mathtt{F} : (x{:}\mathtt{s}), \Pi}{\vdash \mathbf{P}\ x\ \mathtt{o}\text{-}\mathtt{F} : \mathsf{OK}}\ \textit{mode-10}$$

Figure 6. Selected and Simplified Mode Analysis Rules

grounding substitution for free variables in the judgment, and $\mathtt{H} \models^{\Upsilon_I} \sigma(\Delta)$*, then* $\mathtt{H} \models^{\Upsilon_I} \mathtt{F}$.

3.4 Pattern-Matching Algorithm

The pattern-matching algorithm (MP) determines if a given program heap satisfies a formula. We implement MP using an algorithm similar to Prolog's depth-first, bottom-up proof search strategy. When a user-defined predicate is queried, we try all the clause bodies defined for this predicate. In evaluating a clause body, we evaluate the formulas in the body in left-to-right order as they appear. MP takes four arguments: the heap H, a set of locations S that are not usable, a formula F, and a substitution σ for a subset of the free variables in F. It either succeeds and returns a substitution for all the free variables in F, and the locations used in proving F, or fails and returns NO. The set of locations is used to handle linearity and make sure that no piece of the heap is used twice.

For the rest of this section, we will explain the mechanisms that guarantee the termination, correctness, and the memory safety of the pattern-matching algorithm.

3.4.1 Termination Restriction

In order for MP to terminate, we require that some linear resources are consumed when we evaluate the clause body so that the usable heap gets smaller when we call MP on the user-defined predicates in the body. More specifically, in the inductive definitions for predicate $\mathbf{P}$, there has to be at least one clause body that contains only arithmetic formulas and struct predicates, and for clauses whose body contains user-defined predicates, there has to be at least one struct predicate that precedes the first user-defined predicate in the body. We statically check these restrictions at compile time.

3.4.2 Mode Analysis

Our mode analysis serves two important purposes: (1) it ensures MP knows the addresses of data structures it must traverse (*i.e.*, those addresses are ground when they need be), and (2) it ensures these addresses are safe to dereference. In this section, we present selected and simplified rules for the analysis. In particular, we focus on defining two judgments. The first $\vdash \mathbf{P}\ x\ \mathtt{o}{-}\ \mathtt{F}\ \mathsf{OK}$ affirms that the inductive definition ($\mathbf{P}\ x\ \mathtt{o}{-}\ \mathtt{F}$) is well-moded, satisfying both properties (1) and (2) above.[1] The second judgment checks the body of a definition in a context Π, that maps variables to their safety modes s. This second judgment has the form $\Pi \vdash \mathtt{F} : \Pi'$ and affirms that F is well-moded provided that variables in the domain of Π are ground (*i.e.*, will be instantiated and available at run time) and satisfy their associated safety mode. The output context Π' contains the variables that will be ground after the execution of F. Both judgments are parameterized by a function mode that maps each user-defined predicate $\mathbf{P}$ to its declared mode. The rules also use the notation $\Pi \bar{\cup} (x{:}\mathtt{s})$ to denote Π' such that $\Pi'(y) = \Pi(y)$ when $y \neq x$, and $\Pi'(y) = \mathtt{s}'$ when $y = x$ where $\mathtt{s}'$ is the *stronger* of s and $\Pi(x)$. The mode sf is stronger than the mode unsf.

Selected rules from both judgments appear in Figure 6. To understand the difference between predicates with $+$ and $-$ modes, compare rules *mode-1* and *mode-2*. In rule *mode-1*, predicate $\mathbf{P}$ has $+$ mode and hence its argument must be in the input context Π, meaning the argument will have been instantiated and available when execution of MP reaches this point. In contrast, in rule *mode-2*, x need not be in the input context Π, but is added to the output context. Now to understand propagation of safety constraints, compare rules *mode-1* and *mode-3*. In rule *mode-3*, $x{:}\mathtt{s}$ must be in Π since $\mathbf{P}$ still has groundness mode $+$, but since $\mathbf{P}$'s safety mode is unsf, s is unconstrained – it may either be of sf or unsf. A predicate $\mathbf{P}$ that compared its argument to another value but did not dereference it might have the mode shown in rule *mode-3*. Rule *mode-5* shows how mode information is passed left-to-right from one conjunct in a formula to the next.

3.4.3 Formal Results

We have proven several key facts concerning our mode analysis. Our first theorem states that MP terminates if the inductive definitions are well-formed. Judgment $\vdash$ I OK checks the termination constraints correctly.

Theorem 2 (Termination of MP)
If for all $\mathtt{I} \in \Upsilon, \vdash \mathtt{I}\ \mathsf{OK}$*, then* $MP_\Upsilon(\mathtt{H}; S; \mathtt{F}; \sigma)$ *always terminates.*

We have also proven that MP is complete and correct with regard to the semantics if all the user-defined predicates are well-formed.

Theorem 3 (Correctness of MP)
If $\Pi \vdash \mathtt{F} : \Pi'$*, and* $\forall x \in dom(\Pi).\ x \in dom(\sigma)$*, and* $S \subset dom(\mathtt{H})$ *then*

- *either* $\mathtt{MP}(\mathtt{H}; S; \mathtt{F}; \sigma) = (S', \sigma')$ *and* $S' \subset dom(\mathtt{H})$*,* $\sigma \subset \sigma'$*, and* $\mathtt{H}' \models \sigma'(\mathtt{F})$*, and* $dom(\mathtt{H}') = (S' - S)$*,* $\forall x \in dom(\Pi').\ x \in dom(\sigma')$*,*
- *Or* $\mathtt{MP}(\mathtt{H}; S; \mathtt{F}; \sigma) = \mathtt{NO}$*, and there does not exist a heap* $\mathtt{H}'$ *which is the sub-heap of* H *minus the locations in the set* S*, or* σ'*,* $\sigma \subset \sigma'$*, such that* $\mathtt{H}' \models \sigma'(\mathtt{F})$

Finally, we have proven the following theorem stating that MP procedure is memory safe. We say a term tm is well-moded with respect to contexts Π, σ, and heap H, if either tm is an integer, or it is in the domain of σ, and if $\Pi(\mathtt{tm}) = \mathtt{sf}$ then $\sigma(\mathtt{tm})$ is a valid pointer on the heap H.

Theorem 4 (Safety of MP)
If for all $\mathtt{I} \in \Upsilon$*,* I *is well-moded, complies with the termination constraints, and* $\mathbf{P}$ *is a closed shape, and* $\mathtt{H}_1 \models \mathbf{P}(l)$*,* $\Pi \vdash \mathtt{F} : \Pi'$*, and* $\forall \mathtt{tm} \in dom(\Pi)$*.* tm *is well-moded with respect to* Π*,* σ *and* $\mathtt{H}_1$ *then*

- *either* $\mathtt{MP}(\Upsilon)(\mathtt{H}_1 \uplus \mathtt{H}_2; S; \mathtt{F}; \sigma) = (S', \sigma')$*, and* MP *is memory safe, and* $\forall \mathtt{tm} \in dom(\Pi')$*,* tm *is well-moded with respect to* Π'*,* σ' *and* $\mathtt{H}_1$*.*
- *Or* $\mathtt{MP}(\Upsilon)(\mathtt{H}_1 \uplus \mathtt{H}_2; S; \mathtt{F}; \sigma) = \mathtt{NO}$*, and* MP *is memory safe.*

[1] For expository purpose, we only explain the rules for predicates with a single argument. We have proven correct the appropriate generalizations involving multiple arguments.

Basic Types	$\mathbf{t}$	$::=$	$\mathbf{int} \mid \mathbf{ptr}(\mathbf{P})$
Regular Types	τ	$::=$	$\mathbf{t} \mid \mathbf{P}$
Fun Types	τ_f	$::=$	$(\tau_1 \times \cdots \times \tau_n) \rightarrow \mathbf{P}$
Vars	var	$::=$	$\$x \mid x$
Exprs	e	$::=$	$\mathtt{var} \mid n \mid e + e \mid -e$
Args	arg	$::=$	$e \mid \$s$
Shape Forms	Shp	$::=$	root v, F
Shape Patterns	pat	$::=$	: [Shp] | ?[Shp]
Atoms	a	$::=$	A | $\$s$ pat
Conj Clauses	cc	$::=$	a | cc, cc
Branch	b	$::=$	(pat $\rightarrow$ stmt)
Branches	bs	$::=$	b | b $'\mid'$ bs
Statements	stmt	$::=$	skip | $\mathtt{stmt}_1$; $\mathtt{stmt}_2$ | $\$x := e$ | if cc then $\mathtt{stmt}_1$ else $\mathtt{stmt}_2$ | while cc do stmt | switch $\$s$ of bs | $\$s := \{\overline{x{:}\mathbf{t}}\}$ [Shp] | free v | $\$s := f$ ($\overline{\mathtt{arg}}$)
Fun Bodies	fb	$::=$	$\mathtt{stmt}_1$; fb | return $\$s$
Local Decl	ldecl	$::=$	$\mathbf{t}\ \$x := v \mid \mathbf{P}\ \$s$
Fun Decl	fdecl	$::=$	$\mathbf{P}\ f(x_1 : \tau_1, \cdots x_n : \tau_n)\ \{\ \overline{\mathtt{ldecl}};\ \mathtt{fb}\ \}$
Program	prog	$::=$	$\overline{\mathtt{SS}};\ \overline{\mathtt{fdecl}}$
Values	v	$::=$	$x \mid n \mid \$s$

Figure 7. Syntax of Language Constructs

$(E; \mathtt{H}; \mathtt{stmt}) \longmapsto (E'; \mathtt{H}'; \mathtt{stmt}')$

free	$(E; \mathtt{H}; \mathtt{free}\ v) \longmapsto (E; \mathtt{H}_1; \mathtt{skip})$ where $\mathtt{H} = \mathtt{H}_1 \uplus \mathtt{H}_2$ and $\mathtt{H}_2(v) = n$, $dom(\mathtt{H}_2) = \{v, v+1, \cdots, v+n\}$
assign-shape	$(E; \mathtt{H}; \$s{:}\mathbf{P} := \{\overline{x}\}[\mathtt{root}\ (v), \mathtt{F}])$ $\longmapsto (E[\$s := v']; \mathtt{H}'; \mathtt{skip})$ where $(\mathtt{H}', v') = \mathtt{CreateShape}(\mathtt{H}, \mathbf{P}, \{\overline{x}\}[\mathtt{root}\ (v), \mathtt{F}])$ $\mathtt{CreateShape}(\mathtt{H}, \mathbf{P}, \{\overline{x}\}(\mathtt{root}\ n, \mathtt{F})) = (\mathtt{H}', n\,[\,\overline{l}\ /\ \overline{x}\,])$ where 1. $k_i = \mathtt{size}(\mathtt{F}, x_i)$ 2. $(\mathtt{H}_1, l_1) = \mathtt{alloc}(\mathtt{H}, k_1), \cdots,$ $(\mathtt{H}_n, l_n) = \mathtt{alloc}(\mathtt{H}_{n-1}, k_n)$ 3. $\mathtt{F}' = \mathtt{F}\,[\,l_1 \cdots l_n\ /\ x_1 \cdots x_n\,]$ 4. $\mathtt{H}' = \mathtt{H}[v+i := v_i]$ for all $(\mathtt{node}\ v\ (v_1 \cdots v_k)) \in \mathtt{F}'$
If-t	$(E; \mathtt{H}; \mathtt{if\ cc\ then\ stmt}_1\ \mathtt{else\ stmt}_2) \longmapsto (E; \mathtt{H}; \sigma(\mathtt{stmt}_1))$ if $[\![\,\mathtt{cc}\,]\!]_E = (\mathtt{F}, \sigma')$ and $\mathtt{MP}(\mathtt{H}; \mathtt{F}; \emptyset; \sigma') = (SL; \sigma)$
If-f	$(E; \mathtt{H}; \mathtt{if}\ cc\ \mathtt{then\ stmt}_1\ \mathtt{else\ stmt}_2) \longmapsto (E; \mathtt{H}; \mathtt{stmt}_2)$ if $[\![\,cc\,]\!]_E = (\mathtt{F}, \sigma)$ and $\mathtt{MP}(\mathtt{H}; \mathtt{F}; \emptyset; \sigma) = \mathtt{NO}$
while-t	$(E; \mathtt{H}; \mathtt{while\ cc\ do\ stmt})$ $\longmapsto (E; \mathtt{H}; (\sigma(\mathtt{stmt}_1)\ ;\ \mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc\ do\ stmt}))$ if $[\![\,\mathtt{cc}\,]\!]_E = (\mathtt{F}, \sigma')$ and $\mathtt{MP}(\mathtt{H}; \mathtt{F}; \emptyset; \sigma') = (SL; \sigma)$
while-f	$(E; \mathtt{H}; \mathtt{while\ cc\ do\ stmt}) \longmapsto (E; \mathtt{H}; \mathtt{skip})$ if $[\![\,\mathtt{cc}\,]\!]_E = (\mathtt{F}, \sigma)$ and $\mathtt{MP}(\mathtt{H}; \mathtt{F}; \emptyset; \sigma) = \mathtt{NO}$

Figure 8. Selected Operational Semantics

Since the program heap often contains many data structures, Theorem 4 takes the frame property into account: MP is safe on a larger heap than the one containing the shape MP is matching. Intuitively, MP is safe because it only follows pointers reachable from the root of a "closed shape". The termination of MP is crucial since the proof is done by induction on the depth of the derivation of MP. If MP did not terminate, then the induction on the depth of the derivation would not be well-founded.

4. The Programming Language

In this section, we explain how to embed the mode analysis and proof system into the type system, and likewise, the pattern-matching algorithm into the operational semantics, and thereby integrate the verification technique into the language. After introducing various syntactic constructs, we define the formal operational semantics and explain the type system of our language.

4.1 Syntax

A summary of the syntax of our core language is shown in Figure 7. The basic types are the integer type and the pointer type. Functions take a tuple of arguments and always returns a shape type.

We use x to range over logical variables in formulas, $\$x$ to range over stack variables, and $\$s$ to range over shape variables. Shape variables live on the stack and store the starting address of data structures allocated on the heap. We use e to denote expressions and arg to denote function arguments which can either be expressions or shape variables. We use a special root predicate to indicate the starting address of the shape. Shape formulas Shp are the multiplicative conjunction of a set of predicates, the first of which is the root predicate. A shape pattern pat can be either a query pattern (? [Shp]) or a deconstructive pattern (:[Shp]). We have seen the deconstructive pattern in the examples in Section 2. When we only traverse, and read from the heap, but don't perform updates, we use the query pattern (? [Shp]). These two patterns are treated the same operationally, but differently in the type system. The conditional expressions in if statements and while loops are composed of a conjunctive clause cc, which describes the arithmetic constraints and the shape patterns of one or more disjoint data structures. cc is the multiplicative conjunction of atoms a, which can either be arithmetic formulas A, or shape patterns ($\$s$ pat).

The statements include skip, statement sequences, expression assignments, if statements, while loops, switch statements, shape assignments, free, and function calls. The switch statement branches on a shape variable against shape patterns.

A function body is a statement followed by a return instruction. A program consists of shape signatures and a list of function declarations. Lastly, the values in our language are integers, variables, and shape variables.

4.2 Operational Semantics

In this section, we define the operational semantics of our language. Most rules are straightforward. We focus on explaining the interesting ones that dereference the heap using pattern-matching procedure or update the heap via logical formulas.

The machine state for evaluating statements other than the function call statement is a tuple: $(E; \mathtt{H}; \mathtt{stmt})$. Environment E maps stack variables to their values. H is the program heap, and stmt is the statement being evaluated. We write $(E; \mathtt{H}; \mathtt{stmt}) \longmapsto (E'; \mathtt{H}'; \mathtt{stmt}')$ to denote the small-step operational semantics for these statements. Figure 8 is a list of selected rules. We write $E(\$x)$ and $E(\$s)$ to denote the value E maps $\$x$ and $\$s$ to.We write $E(F)$ to denote the formula with values substituted for variables.

To deallocate a tuple, programmers supply the free statement with the starting address v of that tuple. The heaplet to be freed is easily identified, since the size of the tuple is stored in v.

The shape assignment statements allow programmers to create data structures. During the execution of a shape assignment statement, the heap is updated according to the shape formulas in the statement. In the end, the root of the new shape is stored in $\$s$. The core procedure is CreateShape, which takes as arguments, the current heap, the shape name $\mathbf{P}$, and the shape formula. It returns the updated heap and the root of the new shape. In order to define this procedure, we define function $\mathtt{size}(\mathtt{F}, x)$ to be the appropriate size of the tuple x points to according to the shape formula F.

When an if statement is evaluated, the pattern-matching procedure is called to check if the conditional expression is true. If MP succeeds and returns a substitution σ, we continue with the evaluation of the true branch with σ applied; otherwise, the false branch is evaluated. Notice that the conditional expression cc is not in the form of a logical formula; therefore, we need to convert cc to its equivalent formula $\mathtt{F}_{cc}$, before invoking the pattern-matching procedure MP. We define $[\![\,cc\,]\!]_E$ to extract $\mathtt{F}_{cc}$ and a substitution σ from cc. Intuitively, $\mathtt{F}_{cc}$ is the conjunction of all the shape formulas with the run-time values substituted for the stack variables and the root predicate dropped. For example, if $E(\$s) = 100$ and

cc is ($s : [root r, node r (d, next), list next]) then logic variable r must be 100 as the clause specifies that r is the "root" the value of $\$s$. Hence the substitution σ is $\{100/r\}$ and $\mathtt{F}_{\mathtt{cc}}$ is (node r (d, next), list next). MP is called with the current program heap, an empty set (all the locations are usable), the formula $\mathtt{F}_{cc}$, and the substitution σ from $[\![\,\mathtt{cc}\,]\!]_E$.

The while loop is very similar to the if statement. If the conditional expression is true then the loop body is evaluated and the loop will be re-entered; otherwise, the loop is exited.

4.3 Type System

As mentioned earlier, our type system is a linear type system. The contexts in the typing judgments not only keep track of the types of the variables, but also describe the current status of program state: what the valid shapes are, and what the structure of the accessible heap is. The contexts used in the typing judgments are listed below.

Variable Ctx	Ω	$::=$	$\cdot \mid \Omega, \mathtt{var}{:}\mathbf{t}$
Initialized Stack Variable Ctx	Γ	$::=$	$\cdot \mid \Gamma, \$s{:}\mathbf{P}$
Uninitialized Shape Variable Ctx	Θ	$::=$	$\cdot \mid \Theta, \$s{:}\mathbf{P}$
Heap Ctx	Δ	$::=$	$\cdot \mid \Delta, \mathtt{Pu} \mid \Delta, \mathtt{Ps}$

Context Ω maps variables to their types. Both Γ and Θ map shape variables to their types. During type checking, Γ specifies the initialized shape variables, while Θ specifies the uninitialized shape variables. Variables in Θ are not currently usable, but may be initialized later in the program. Context Δ is a set of formulas that describe the accessible portions of the heap. Context Γ and Δ describe the entire program heap. For example, if $\Gamma = \$s{:}\mathtt{listshape}$ and Δ = node 400 (11, 0), and the environment is $E = \$s \mapsto 100$, then the current heap must satisfy formula (listshape 100, node 400 (11,0)).
The main judgments in our type system are listed below.

Expression typing	$\Omega \vdash_e e : \mathbf{t}$
Conj Clause typing	$\Omega; \Gamma \vdash_{\mathtt{cc}} \mathtt{cc} : (\Gamma'; \Theta; \Delta)$
Conj Clause Modes checking	$\Pi \vdash \mathtt{cc} : \Pi'$
Statement typing	$\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} : (\Gamma'; \Theta'; \Delta')$

Each of these judgments are also implicitly indexed by the contexts for shape signatures Λ, axioms Υ, and function type bindings Φ. We leave these contexts implicit as they are invariant through the type checking process.

Conjunctive Clause Typing. The typing judgment for conjunctive clauses, which is used in type checking if statements and while loops, has the form $\Omega; \Gamma \vdash_{\mathtt{cc}} \mathtt{cc} : (\Gamma'; \Theta; \Delta)$. The interesting rule is when cc is a deconstructive shape pattern.

$$\frac{\Gamma = \Gamma', \$s{:}\mathbf{P}, \quad \Upsilon; \mathtt{F}_A, \mathtt{F} \Longrightarrow \mathbf{P}(y) \qquad \mathtt{FV(F)} \cap \Omega_{\$s} = \emptyset}{\Omega; \Gamma \vdash \$s : [\mathtt{root}\, y, \mathtt{F}_A, \mathtt{F}] : (\Gamma'; \$s{:}\mathbf{P}; \mathtt{F})}$$

Here, shape variable $\$s$ is deconstructed by shape pattern $(\mathtt{root}\, y, \mathtt{F}_A, \mathtt{F})$, where $\mathtt{F}_A$ is the arithmetic formulas. If pattern matching succeeds, then there exists some substitution σ such that the heaplet pointed to by $\$s$ can be described by $\sigma(\mathtt{F}_A, \mathtt{F})$. Therefore, in the postcondition, F appears in the Δ context providing describing formulas to access the heap $\$s$ points to; shape variable $\$s$ becomes uninitialized, and the type binding of $\$s$ is in the Θ context. The condition that no stack variables appear free in F ensures that the formulas are valid descriptions of the heap regardless of imperative variable assignments. Finally, the logical derivation checks that the shape formulas entails the desired shape. By the soundness of logical deduction, we know that any heaplet H matched by the shape formula also satisfies $\mathbf{P}$ v, where v is the run-time value of $\$s$.

Conjunctive Clause Mode Checking At run time, MP is called on the conjunctive clauses cc. So we have to apply mode analysis on cc to ensure the memory safety of MP. The mode checking for cc uses the mode checking for formulas and treats cc as the multiplicative conjunction of the formulas in each atom in cc. As an example, the rule for checking the deconstructive pattern is shown below.

$$\frac{\Pi \uplus \{x{:}\mathtt{sf}\} \vdash \mathtt{F} : \Pi'}{\Pi \vdash \$s : [\mathtt{root}\, x, \mathtt{F}] : \Pi'}$$

Since $\$s$ points to a valid shape, its root pointer is a valid pointer. Therefore the argument of the root predicate is added as a safe pointer argument in the ground context Π while checking the formula in the shape pattern.

Statement Type Checking. The typing judgment for statements has the form $\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} : (\Gamma'; \Theta'; \Delta')$. A selected set of typing rules is listed in Figure 9.

The rule for if statements first infers the types of the free variables (we omit the definition of infer but it is straight-forward). Then we type check the conjunctive clause cc. The true branch is taken when cc is proven to be true, and at that point the describing formulas from examining cc are proven to be valid; hence the true branch is checked under the new state resulting from checking cc. The false branch is checked under the original state. The end of the if statement is a program merge point, so the true and the false branch lead to the same state. The mode checking of cc guarantees that when MP is called on cc it won't access dangling pointers. The initial Π context contains groundness and safety properties of the arguments when cc is evaluated, and it depends on the variable context Ω. Before evaluating a statement, the run-time values should already have been substituted for the bound variables. Therefore, all the variables in Ω are ground before we evaluate cc. We have no information on the validity of the pointer variables, so they are considered unsafe. $\mathtt{ground}(\Omega)$ is defined below.

$$\mathtt{ground}(\Omega) = \{\mathtt{var} \mid \Omega(\mathtt{var}) = \mathbf{int}\} \cup \{(\mathtt{var}{:}\mathtt{unsf}) \mid \Omega(\mathtt{var}) = \mathbf{ptr}(\mathbf{P})\}$$

Since the only safe pointers we assume before evaluating cc are the root pointers of valid shapes, the memory safety of MP when evaluating cc is guaranteed through the safety of MP (Theorem 4).

While loops are similar to if statements. After type checking the conjunctive clause, the loop body is checked against the new states. The resulting states should be the same as the original states, so that the loop can be re-entered. This means that the states under which the while loop is type checked are in effect loop invariants.

The rule for shape assignment first checks that the shape variable $\$s$ is uninitialized – this check prevents memory leaks. It does so by checking that $\$s$ belongs to Θ (not Γ). The third premise in the assignment rule checks that the shape formula entails the shape of the variable $\$s$. The rest of the premises entail that the union of the formulas used to construct this shape and the leftover formulas in Δ' should be the same as the formulas given at the beginning in Δ plus the new heaplets allocated. It looks complicated because we allow multiple updates to the heap during assignment. For example, if node l (5,0) is available, then we allow node l (10, 0) to be used in the shape assignment. This means that the heap cell that used to contain 5 now contains 10.

The rule for free checks that the location to be freed is among the accessible portion of the heap. After freeing, the formula describing the freed heaplet is deleted from the context, and can never be accessed again.

4.4 Type Safety

The machine state for evaluating function bodies requires an additional control stack S, which is a stack of evaluation contexts waiting for the return of function calls. The typing judgment for machine state has the form $\vdash (Es; \mathtt{H}; S; \mathtt{fb})\,\mathsf{OK}$. We proved the following type-safety theorem for our language.

Theorem 5 (Type Safety)
if $\vdash (E; \mathtt{H}; S; \mathtt{fb})\,\mathsf{OK}$ *then either* $(E; \mathtt{H}; S; \mathtt{fb}) = (\bullet; \mathtt{H}; \bullet; \mathtt{halt})$
or exists E', $\mathtt{H}'$, S', $\mathtt{fb}'$ *such that* $(E; \mathtt{H}; S; \mathtt{fb}) \longmapsto (E'; \mathtt{H}'; S'; \mathtt{fb}')$ *and* $\vdash (E'; \mathtt{H}'; S'; \mathtt{fb}')\,\mathsf{OK}$

$$\frac{\Omega' = \texttt{infer(cc)} \quad \Omega', \Omega; \Gamma \vdash \texttt{cc} : (\Gamma', \Theta', \Delta') \quad \Omega', \Omega; \Gamma'; \Theta, \Theta'; \Delta, \Delta' \vdash \texttt{stmt}_1 : (\Gamma''; \Theta''; \Delta'') \quad \Omega; \Gamma; \Theta; \Delta \vdash \texttt{stmt}_2 : (\Gamma''; \Theta''; \Delta'') \quad \Pi = \texttt{ground}(\Omega) \quad \Pi \vdash \texttt{cc} : \Pi'}{\Omega; \Gamma; \Theta; \Delta \vdash \texttt{if cc then stmt}_1 \texttt{ else stmt}_2 : (\Gamma''; \Theta''; \Delta'')} \textit{ if}$$

$$\frac{\Omega' = \texttt{infer(cc)} \quad \Omega', \Omega; \Gamma \vdash \texttt{cc} : (\Gamma', \Theta', \Delta') \quad \Omega', \Omega; \Gamma'; \Theta, \Theta'; \Delta, \Delta' \vdash \texttt{stmt} : (\Gamma; \Theta; \Delta) \quad \Pi = \texttt{ground}(\Omega) \quad \Pi \vdash \texttt{cc} : \Pi'}{\Omega; \Gamma; \Theta; \Delta \vdash \texttt{while cc do stmt} : (\Gamma; \Theta; \Delta)} \textit{ while}$$

$$\frac{\Omega' = \texttt{infer}(\overline{x}, \texttt{F}) \quad \Theta = \Theta', (\$s : \mathbf{P}) \quad \Upsilon; \texttt{F} \Longrightarrow \mathbf{P}(v) \quad \Delta_x = \{\texttt{Ps}\ x_i\ \overline{e} \mid \texttt{Ps}\ x_i\ \overline{e} \in \texttt{F}\} \quad \Delta = \Delta', \Delta'' \quad \texttt{F} = \Delta_x, \Delta_F \quad \forall \texttt{Pu}, \texttt{Pu} \in \Delta'' \text{ iff } \texttt{Pu} \in \Delta_F \quad \forall \texttt{Ps tm}\ \overline{e} \in \Delta'' \text{ iff } \texttt{Ps tm}\ \overline{e}' \in \Delta_F}{\Omega; \Gamma; \Theta; \Delta \vdash \$s{:}\mathbf{P} \ := \ \{\overline{x}\}[\texttt{root}\,(v), \texttt{F}] : (\Gamma', \$s{:}\mathbf{P}; \Theta'; \Delta')} \textit{ assign-shape}$$

$$\frac{\Delta = (\texttt{Ps}\ v\ e_1 \cdots e_k), \Delta'}{\Omega; \Gamma; \Theta; \Delta \vdash \texttt{free}\ (v) : (\Gamma; \Theta; \Delta')} \textit{ free}$$

Figure 9. Selected Statement Typing Rules

5. A Further Example

To demonstrate the expressiveness of our language, we coded the shapes and operations of various commonly used data structures such as singly/doubly linked circular/non-circular lists, binary trees, and graphs represented as adjacency lists. In this section, we will explain how to define adjacency lists, the most interesting and complex of the data structures we have studied.

In Figure 10, the data structure on the left is an adjacency list representation of the directed graph on the right. Each node in the graph is represented as a tuple composed of three fields. The first field is a data field; the second field is a pointer to another node in the graph, and the last field is a pointer to the adjacency list of outgoing edges. In this example, node a is represented by the tuple starting at address 100, b is represented by the one starting at 200, and c is represented by the one starting at 300. In this data structure, each tuple representing a graph node is pointed to from the previous node, and from the node's incoming edges. For example, node c (tuple starting at address 300) is pointed to by the next pointer from node b, and from the adjacency list of both node a and b.

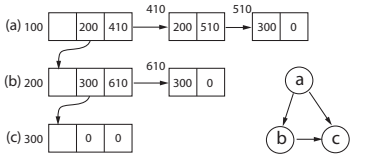

Figure 10. An adjacency list.

The Shape Signature. We present the key parts of the shape signature of an adjacency list in Figure 11. The tuples representing nodes have a different size than the tuples representing edges; hence we need two struct predicates: `graphnode` to store node information, and `adjnode` to store edge information.

Beginning at the bottom of Figure 11, we see the definition of predicate `adjlist X B`. This predicate describes the list of outgoing edges from a node. The argument `X` points to the beginning of the list and the argument `B` represents the set of nodes contained therein. Since definitions involving sets appear frequently, our language includes built-in support for them. In particular [] denotes the empty set, [n] denotes a singleton set, s1 U s2 denotes the union of two sets and `s1<=s2` denotes the subset relation. The definition of a adjlist shows how to use this notation to build an edge list together with a set representing its contents.

The predicate `(nodelist X A B)` is valid when `X` points to a graph data structure in which `A` is the complete set of graph nodes

```
1    graphshape{
2     struct graphnode: ...
3     struct adjnode: ...
     ...
10     graphshape X o- graph X.
11     graph X o- nodelist X A B, B <= A.
12     nodelist X A B o- X = 0, A = [], B = [];
13                  graphnode X <d, next, adjl>,
14                  adjlist adjl G, nodelist next A1 B1,
15                  A = [X] U A1, B = B1 U G.
16     adjlist X B o- X = 0, B = [];
17                    adjnode X <n, next>,
18                    adjlist next B1, B = [n] U B1.
   ... }
```

Figure 11. Shape Signature for Graph

and `B` is the subset of nodes that have at least one incoming edge. For example, the adjacency list in Figure 10 can be described by `adjlist 100 [100,200,300] [200,300]`. The base case for the definition of `nodelist` is trivial. In the second case, `X` is a graph node that has some data `d`, a pointer `next` pointing to the next graph node (`nodelist next A1 B1`), and a pointer `adjl` pointing to the outgoing edges of `X` (`adjlist adjl G`). The set of graph nodes is the union of `A1` and `[X]`, and the set of nodes that has at lease one incoming edges is the union of `G` and `B1`.

Predicate `graph X` is defined in terms of predicate `nodelist`. `X` points to an adjacency list representation of a graph if `X` points to a `nodelist` and all the edges point to valid graph nodes (`A <= B`). This last constraints guarantees that one cannot reach dangling pointers while traversing the graph.

Graph Operations. We have coded and verified the most important operations on graphs, including insertion and deletion of both nodes and edges. Space constraints prevent us from presenting the complete examples here. However, we will remark that the most interesting operation is node deletion, since nodes may be pointed to by arbitrarily many edges. Properly deleting a node requires first that all edges pointing to it are deleted. When no edges remain, the node n to be deleted appears in set `A` but not set `B` of a valid `(nodelist X A B)` predicate. This fact provides sufficient information that one can then delete the node and reform the graph, confident it contains no dangling pointers. The type system verifies this last step of the algorithm is indeed correct. Alternatively, if programmers attempt to delete a node while edges remain pointing to it, they will find it impossible to convince the system that the remaining data structure satisfies the definition of a graph. In particular, the set constraint on line 11 of Figure 11, (`B <= A`) will fail.

6. Related Work

Several researchers have used declarative specifications of complex data structures to generate code and implement pattern-matching operations. For example, Klarlund and Schwartzbach used 2nd-order monadic logic to describe *graph types*, a generalization of ML-style data types [13]. Similarly, Fradet and Le Métayer developed *shape types* [5] by using context-free graph grammars. Both of these works were highly inspirational to us. However, space reserved for one of Klarlund's graph types can not be reused in construction of another type, nor can graph types be deallocated. Fradet's shape types, while interesting, did not come with a facility for expressing relations between different shapes similar to our axioms, and consequently it appears that they cannot be used effectively inside while loops or other looping constructs. Perhaps more important than the differences in expressive power, is the fact that our language has the promise of synergy with new verification techniques based on substructural logics and with modern type systems for resource control, including those in Vault [3] and Cyclone [7, 9].

More generally, there are many, many different varieties of static analysis aimed at verifying programs that manipulate pointers. Some of them use logical techniques and some of them do not. These analysis range from standard alias analysis to data flow analysis to abstract interpretation to model checking to shape analysis (see Sagiv et al.'s work [21], for example). We distinguish ourselves from this large and important volume of work by noting that we do not verify low-level statements that explicitly dereference pointers. Instead, we aim to replace low-level pointer manipulation, which requires verification, with higher-level data structure specification, which is "correct by construction."

As noted in the introduction, we follow in the intellectual footsteps of O'Hearn, Reynolds, Yang and others, who have developed the theory and implementation of separation logic and used it to verify low-level pointer programs [20, 11, 12]. However, we chose to pursue our research starting with a foundation in linear logic as opposed to the logic of bunched implications, which underlies separation logic. One motivating factor for doing so was the presence of readily available linear logic programming languages [10, 14] and automated theorem provers [2, 15], which we have used to experiment with ideas and to implement a prototype for our language.

The fragment of linear logic that we choose to use as the base logic to describe shapes has the same no-weakening and no-contraction properties as the multiplicative fragment of separation logic (linear logic and the logic of bunched implications [17], the basis for separation logic coincide exactly on this fragment). We call our logic "linear" since its proof theory uses two contexts (one the linear and one unrestricted) and hence it shares the same structure as Girard's work [6]. Bunched implications and separation logic have an additive implication and an additive conjunction, which do not appear in our logic. We can simulate the additives when they are used to manipulate "pure formulas" (those formulas and that do not refer to the heap), but not when they are used to describe storage (which can be useful to describe certain aliasing patterns). In the future, we plan to explore extending the system with either linear logic's additive conjunction or related ideas found in linear type systems [23, 3, 22, 25, 8, 16, 26].

7. Conclusion

We have developed a new programming paradigm that uses linear logical formulas as specifications for defining and manipulating heap-allocated recursive data structures. A key component of the new system is an algorithm for heap-shape pattern matching, derived in part from an understanding of the operation of linear logic programming languages. To ensure the safety of pattern matching, we extended the mode analysis found in many logic programming languages to check for dangling pointers. Lastly, we integrated all these new ideas into an imperative programming language, for which we have developed a prototype interpreter and type checker. Our new language will facilitate safe construction, deconstruction and deallocation of sophisticated heap-allocated data structures.

References

[1] 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.

[2] K. Chaudhuri and F. Pfenning. A focusing inverse method prover for first-order linear logic. In *20th International Conference on Automated Deduction (CADE-20)*, July 2005.

[3] R. Deline and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM Press.

[4] M. Fähndrich and R. Deline. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, June 2002.

[5] P. Fradet and D. L. Métayer. Shape types. In *POPL '97*, 1997.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, 2003.

[8] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, 2002.

[9] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *International Symposium on Memory Management*, pages 73–84, Oct. 2004.

[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, 1994.

[11] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, Jan. 2001.

[12] C. C. Josh Berdine and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[13] N. Klarlund and M. Schwartzbach. Graph types. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 196–205, Charleston, Jan. 1993.

[14] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *PPDP '05*, 2005.

[15] H. Mantel and J. Otten. linTAP: A tableau prover for linear logic. *Lecture Notes in Computer Science*, 1617, 1999.

[16] 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.

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

[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] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, 2004.

[20] 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.

[21] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, Jan. 1999.

[22] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, Mar. 2000.

[23] D. Walker. *Substructural Type Systems*, chapter 1. MIT Press, 2005.

[24] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, May 2000.

[25] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, Sept. 2000.

[26] D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*. Springer-Verlag LNCS vol. 3350, January 2005.