Linear Logic, Heap-shape Patterns and Imperative Programming

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.

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 [19, 12, 17]. 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 [19]. These authors specify heap-shape invariants using a variant of the logic of bunched implications (BI) [18]. They then include the BI specifications in a Hoare logic, which they use 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).[1] 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 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.

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. Finally, in the last two sections, we discuss related and future work.

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 struc-

[1] The differences between our logic, which we call "linear," due to its dual-zone (unrestricted and linear) proof theory, and the corresponding fragment of O'Hearn's separation logic appear largely cosmetic.

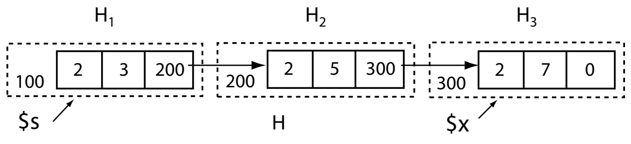

Figure 1. Memory containing a linked list.

tures. Compared to C, where the type definition for a singly linked list is no different from that of a circular list, our language allows programmers to capture aliasing constraints and shape invariants of the recursive data structures.

In this section, first, we introduce the program heap and its basic describing formulas. 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 explain the invariants that keep our system memory safe. We only give a brief introduction to our system, but defer precise explanation of the technical details to later in the paper.

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.

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 `struct x T`, where `x` is the starting address of the tuple and `T` is the tuple of heap contents. For example, `(struct 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 `(struct 100 (3, 200), struct 200 (5, 300), struct 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 equality and inequality constraints over integers.

In addition to using these primitive descriptions, programmers can create new definitions to describe heap 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 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 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, 15]. Hence, 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-," and 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); (struct 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 ","; 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 a more complex 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), struct 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 makes use of the `listseg` predicate to define a queue.

```
queue X Y o-
  ((X = 0), (Y = 0));
  (listseg X Y, struct 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 line describes the case where 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 100, and tail pointer is 300 (`queue 100 300`).

Defining tree-shaped data is no more difficult than defining list-shaped data. As an example, consider the following binary tree definition.

```
btree X o-
  (X = 0);
  (struct 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 shape patterns, such as `(listseg x y, list y)`, have the desired shapes (`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 built-in and user-defined predicates are given both types and modes. The purpose of the types should be familiar to all programmers: they 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 effective.

To understand the purpose of mode declarations and mode analysis, consider the problem of matching the predicate `struct 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 $\bar{H}$`(l) = (77,34)`. However, without any additional information, it would seem the only possible algorithm would involve examining the contents of every address in `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 (`struct l' (D,X), struct X (77,34)`) against some heap `H`. Can we do it? Yes. Simply look up `l'` in `H`. Doing so will determine precise values for `D` and `X`. The value of `X` subsequently used to determine whether $\bar{H}$`(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 `struct X (...)` in constant time, we give the first position (`X`) the input mode (`+`). Once the first argument of the `struct` 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 `struct l (...)` against a heap `H`, `l` may be ground, but not necessarily a valid address in H. A second component of our mode analysis characterizes pointers as either `yes` (definitely not dangling) or `no` (possibly dangling) and thereby helps guarantee the matching algorithm does not go wrong. In general, we use `s` to range over either `yes` or `no` modes.

```
listshape {
 struct    : (+,yes,yes) ptr(listshape)
         -> - int
         -> (-,yes,yes) ptr(listshape)
         -> o.

 listshape : (+,yes,yes) ptr(listshape) -> o.
 list      : (+,yes,yes) ptr(listshape) -> o.
 listseg   : (+,yes,yes) ptr(listshape)
         -> (+,no,yes) ptr(listshape)
         -> o.

 listshape X o- list X.

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

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

Figure 2. Singly Linked List Shape Signature

The complete mode for arguments of pointer type is a tuple $(\mathtt{g}, \mathtt{s}_{in}, \mathtt{s}_{out})$, where `g` describes the argument's groundness property, and $\mathtt{s}_{in}$ and $\mathtt{s}_{out}$ describe its safety property at the time just prior to execution of the predicate and just after execution of the predicate respectively. Integers are not dereferenced and hence their modes consist exclusively of the groundness condition `g`.

Only certain combinations of the pointer's tuple of modes are valid. When `g` is don't care (`*`), the safety properties are not relevant, so we set both $\mathtt{s}_{in}$ and $\mathtt{s}_{out}$ to `no`. When `g` is an output, only $\mathtt{s}_{out}$ is relevant (as the pointer's value isn't known on input). When `g` is an input, the pair $\mathtt{s}_{in}$ and $\mathtt{s}_{out}$ can either both be `yes`, both be `no`, or be (`no`,`yes`). The last possibility being a situation in which the evaluation of the predicate has allowed us to learn that a particular pointer is safe. 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 : (+,yes,yes) ptr(listshape) -> o.
```

Putting the Declarations Together. Figure 2 is the full shape signature for `listshape`. It contains the definition of `list` and `listseg` as well as a top-level definition for the overall structure called `listshape`. The axioms are separated from the inductive definitions by the keyword `with`.

The Meanings of the Logical Rules The shape signature essentially defines a linear logic program. Just as a logic programming language has a three-fold interpretations of the meanings of the rules [24, 14]— proof-theoretic, model-theoretic, computational definitions—the shape signature also defines a proof-theoretic meaning which is used during type checking, a model-theoretic meaning which is used to capture the memory-safety invariants preserved through execution, and a computational definitions meaning used to derive a pattern-matching algorithm at run time.

2.4 The Programming Language

In the previous subsection, we showed how to define data structure shapes using linear logic; in this section we will explain how to incorporate these logical definitions 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 **ptr**(`listshape`), specify the specific shape 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:= [root a1,
      struct a1 (3, a2),
      struct a2 (5, a3),
      struct 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).

Assuming variables `a1`, `a2`, and `a3` are not currently in scope at the site of this assignment, the system must find values to bind to them that will make the right-hand side a valid `listshape`. The system achieves this goal by allocating three new tuples of storage. The size of each tuple is determined by examining the type declaration of the `struct` predicate in the shape signature. Each variable is subsequently bound to the address of the corresponding tuple.[2] In general, a shape assignment will always allocate new space in this way in order to generate locations to bind to fresh pointer variables.

Once space has been allocated, the integer data fields are initialized with the constant values appearing in the shape description. Finally, the location specified by the `root` predicate is stored into the imperative variable $\$s$. This special `root` predicate must always appear in the shape on the right-hand side of a shape assignment.

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, struct 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 $\$s$, `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 the revealing that the heaplet in question satisfies the following elaborate formula:

```
∃r:ptr(listshape).∃d:int.∃next:ptr(listshape).
  (root r, struct r (d, next), list next)
```

Pattern matching normally occurs in the context of an if statement or a while loop in our language. Here is an example in the context of an if statement.

```
if $s:[root r, struct 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 the conditional expression. If the conditional is true, 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 conditional expression is false, then the false branch is taken. The variables in the condition are not in scope in the false branch. Suppose $\$s$ points to the first tuple in the `H` displayed in Figure 1. When the conditional expression is evaluated, `r` will be bound to 100, `d` will be bound to 3, and `next` will be bound to 200, and execution will proceed with evaluation of the true branch. In the true branch, we free the first tuple of the list, then reconstruct a list using the rest of the old list. The predicate `root next` indicates that the root of this new shape is `next`. Operationally, the run-time value of `next`, 200, is stored in the variable $\$s$.

2.5 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 memory, in much the same way as O'Hearn's separation logic or its closely related type systems [26, 27, 4, 1, 28]. 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. In our system specifically, when a data structure is deconstructed via a shape pattern, the formula describing the pattern appears in the typing context and describes the accessible portion of memory. Moreover, heap location v can be freed only if formula (`struct v` $\overline{\texttt{tm}}$) appears in the typing context, which means that the tuple starting at v is a valid heaplet of the program heap. After v is freed, the predicate is deleted, and the linear constraints guarantee v can never again be accessed.

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 program:

[2] In our formal presentation, we explicitly quantify over the shape's free variables. We omit the variable bindings in our examples as they may be inferred using the type of the imperative variable being assigned and its shape signature. Any free variables may be assumed to be bound at the top-level.

```
1    ptr(listshape) $p := 0;
...
2    if $s: [root r, struct r (d, next), list next]
3    then $p := r;
4         free r;
5         $s := [root next, list next]
6    else print ‘‘list is empty’’
7    if $s: [root r1, struct $p (d1, next1),
8            listseg r1 $p,  list next1]
9    then ···
10   else ···
```

The first if statement on line 2 frees the head of the list and reassembles the shape using the rest of the list. Notice that stack variable $\$p$ is assigned the value of the freed location. This assignment itself is legal. However, the shape pattern in the second if statement will go ahead and dereference the location stored in `$p`, which was already freed. The solution is to use mode and type analysis to make sure that predicate `struct` requires a valid run-time pointer as its first argument. We can obtain valid run-time pointers from the set of locations reachable from the root of a well-formed shape.

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 other like it.

2.6 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. Like 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 others relating to trees that do not).

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 non-deterministic.

Term	$\mathtt{tm}$	$::=$	$x \mid n \mid -\mathtt{tm} \mid \mathtt{tm} + \mathtt{tm}$
Arith Pred	$\mathtt{Pa}$	$::=$	$\mathtt{tm} = \mathtt{tm} \mid \mathtt{tm} > \mathtt{tm}$
Arith Formula	$\mathtt{A}$	$::=$	$\mathtt{Pa} \mid \mathtt{not}\ \mathtt{Pa}$
State Pred	$\mathtt{Ps}$	$::=$	$\mathtt{struct}\ \mathtt{tm}_1\ \overline{\mathtt{tm}}$
User-Defined Pred	$\mathtt{Pu}$	$::=$	$\mathbf{P}\ \overline{\mathtt{tm}}$
Literals	$\mathtt{L}$	$::=$	$\mathtt{A} \mid \mathtt{Ps} \mid \mathbf{P}$
Formulas	$\mathtt{F}$	$::=$	$\mathtt{emp} \mid \mathtt{L}, \mathtt{F}$
Inductive Def	$\mathtt{I}$	$::=$	$(\overline{\mathtt{F}} \multimap \mathtt{Pu})$
Axiom	$\mathtt{Ax}$	$::=$	$(\mathtt{F} \multimap \mathtt{Pu})$
Groundness	$\mathtt{g}$	$::=$	$+ \mid - \mid *$
Safety Qualifier	$\mathtt{s}$	$::=$	$\mathtt{yes} \mid \mathtt{no}$
Mode	$\mathtt{m}$	$::=$	$\mathtt{g} \mid (\mathtt{g}, \mathtt{s}_{in}, \mathtt{s}_{out})$
Arg Type	$\mathtt{argtp}$	$::=$	$\mathtt{g}\ \mathbf{int} \mid (\mathtt{g}, \mathtt{s}_{in}, \mathtt{s}_{out})\ \mathbf{ptr}(\mathbf{P})$
Pred Type	$\mathtt{pt}$	$::=$	$\mathtt{o} \mid \mathtt{argtp} \to \mathtt{pt}$
Pred Type Decl	$\mathtt{pdecl}$	$::=$	$\mathbf{P}\ :\ \mathtt{pt}$
Shape Signature	$\mathtt{SS}$	$::=$	$\mathbf{P}\{\ \overline{\mathtt{pdecl}};\ (\mathtt{F}\ \multimap\ \mathbf{P}\ x).\ \overline{\mathtt{I}}.\ \overline{\mathtt{Ax}}\}$
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 3. Syntax of Logical Constructs

Again, programmers must verify this property themselves by hand. Once again, it holds for all shapes described in this paper.

These requirements are not surprising. In order to reason about loops, separation logic also contains an axiom which is the same as the one relating `list` and `listseg` in our system. The soundness of that particular axiom is required for the soundness of all the work on verification using separation logic. The uniqueness of shapes requirement is very similar to the *precise predicate* in the work on separation and information hiding [20]. These requirements could well be the common invariants these logical systems should have when used in program verification.

One of the strength of our framework is that we identified all the constraints on the logical definitions for the whole system to be sound. Programmers can supply any definition that complies with the constraints and obtain a sound system.

In the future, we would like to devise mechanisms to check these properties automatically. If we succeed, the result is likely to contribute to the automatic checking of those constraints for separation logic as well. For now, we view them as orthogonal to the definition of the rest of the system.

3. Logical Shape Signatures

In this section, we first introduce the syntactic constructs for defining shape signatures. Next, we explain the logical deduction rules and the semantics generated from the signatures. Finally, we present the pattern-matching procedure and the mode and type checking rules.

3.1 Syntax

The syntactic constructs of our logic are listed in Figure 3. Throughout the paper we use the overbar notation $\overline{x}$ to denote a vector of the constructs under the bar. We use `tm` to range over terms. The arithmetic predicates are the equality and partial order of terms. Arithmetic formulas include arithmetic predicates and their negations. The state predicate `struct tm` $\overline{\mathtt{tm}}$ describes a heaplet starting at `tm` with contents $\overline{\mathtt{tm}}$. In previous examples, we used tuples to describe the contents of the heap. To simply the core language, in the formal syntax, we use a curried style instead. We use $\mathbf{P}$ 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 a 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.

- $\mathtt{H} \vDash \mathtt{emp}$ iff $\mathtt{H} = \emptyset$.
- $\mathtt{H} \vDash \mathtt{A}$ iff $\mathtt{H} = \emptyset$, and $\mathtt{A}$ is a valid arithmetic formula.
- $\mathtt{H} \vDash \mathtt{struct}\ v\, v_1 \cdots v_n$ 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 quantifi ed.
- $\mathtt{H} \vDash \mathbf{P}^n\ \overline{v}$ iff $\Upsilon(\mathbf{P}) = \mathbf{P}\ \overline{x}$ o- $\overline{\mathtt{F}}$, and $\exists i$ st. $\exists \mathbf{P}' \in dom(\mathtt{F}_i)$. $\mathtt{H} \vDash \mathtt{F}_i\,[\,\overline{v}\,/\,\overline{x}\,]$ and $n-1$ is the maximum of the index number of $\mathbf{P}'$ in $\mathtt{F}_i$.

Figure 4. Selected Rules of the Semantics for Formulas

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 clause bodies of the same predicate into one definition $\mathtt{I}$. The notation $\overline{\mathtt{F}}$ means the additive disjunction of all the $\mathtt{F}_i$ in $\overline{\mathtt{F}}$. Axioms are also clauses. They have very a different role in generating proof system and semantics than the inductive definitions.

The argument type of a predicate is the mode followed by the type of the argument. A fully applied predicate has type $\mathtt{o}$.

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 present the formal definition of the store semantics of our logic in Figure 4. We use $\mathtt{H} \vDash^{\Upsilon} \mathtt{F}$ to mean that heap $\mathtt{H}$ can be described by formula $\mathtt{F}$, 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 $\mathbf{P}$. The semantics of arithmetic formulas require the heap to be empty. Heap $\mathtt{H}$ satisfies $\mathtt{struct\ v\ v_1\ \cdots\ v_n}$, if the domain of $\mathtt{H}$ contains exactly the $n+1$ consecutive locations starting from $\mathtt{v}$, and the first location of $\mathtt{H}$ stores the size of the tuple, and values $\mathtt{v_1}$ through $\mathtt{v_n}$ match the contents of the heap.

For the user-defined predicates, we use an index number n to indicate the number of unrollings of the inductive definitions. This index number is closely related to the size of the heap. For example, the heaplet $\mathtt{H}_3$, $(\mathtt{H}_2 \uplus \mathtt{H}_3)$, $\mathtt{H}$ in Figure 1 can be described by $\mathtt{list}^1\ \mathtt{300}$, $\mathtt{list}^2\ \mathtt{200}$, $\mathtt{list}^3\ \mathtt{100}$ respectively. When a clause body is composed of only the arithmetic formulas and the state formulas, the index number is 0. This is the base case where we start to build a recursive data structure. For $\mathtt{list\ x}$, $\mathtt{list}^0\ \mathtt{0}$ is the base case. For the inductive case, the index number of $\mathtt{Pu}$ is the maximum of the index numbers for formulas in its body increased by 1, since the body is the one step unrolling of the head. This is the case where we build up larger data structures from smaller ones. Notice that the semantics does not reference the axioms and is solely determined by the inductive definitions.

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: $\Omega \,|\, \Gamma; \Delta \Longrightarrow \mathtt{F}$. Context Ω contains the free variables in this judgment. Γ is the unrestricted context (hypotheses may be used any number of times) and Δ is the linear context. Initially, Γ will be populated by the inductive definitions and axioms from the shape signatures.

One important property of the proof system is the soundness of logical deduction. We proved that our logical deduction system

$$\frac{\begin{array}{c}\sigma(\mathtt{tm}) \neq 0 \text{ and } \sigma(\mathtt{tm}) \notin S \qquad \bar{\mathtt{H}}(\sigma(\mathtt{tm})) = (v_1, \cdots, v_n)\\ \mathtt{MP}(\mathtt{H}; S; \mathtt{tm}_1 = v_1; \sigma) = (S, \sigma_1)\\ \cdots\\ \mathtt{MP}(\mathtt{H}; S; \mathtt{tm}_n = v_n; \sigma_{n-1}) = (S, \sigma_n)\end{array}}{\mathtt{MP}(\mathtt{H}; S; \mathtt{struct}\ \mathtt{tm}\ \mathtt{tm}_1 \cdots \mathtt{tm}_n; \sigma) = (S \cup \{\sigma(x), \cdots, \sigma(x)+n\}, \sigma_n)}$$

$$\frac{\Upsilon(\mathbf{P}) = (\overline{\mathtt{F}} \multimap \mathbf{P}\ \overline{y}) \quad \forall i \in [1,k],\ \mathtt{MP}(\mathtt{H}; S; \mathtt{F}_i\,[\,\overline{\mathtt{tm}}\,/\,\overline{y}\,]; \sigma) = \mathtt{NO}}{\mathtt{MP}(\mathtt{H}; S; \mathbf{P}\ \overline{\mathtt{tm}}; \sigma) = \mathtt{NO}}$$

$$\frac{\Upsilon(\mathbf{P}) = (\overline{\mathtt{F}} \multimap \mathbf{P}\ \overline{y}) \quad \exists i \in [1,k],\ \mathtt{MP}(\mathtt{H}; S; \mathtt{F}_i\,[\,\overline{\mathtt{tm}}\,/\,\overline{y}\,]; \sigma) = (S_i, \sigma_i)}{\mathtt{MP}(\mathtt{H}; S; \mathbf{P}\ \overline{\mathtt{tm}}; \sigma) = (S_i, \sigma_i)}$$

$$\frac{\mathtt{MP}(\mathtt{H}; S; \mathtt{L}; \sigma) = \mathtt{NO}}{\mathtt{MP}(\mathtt{H}; S; (\mathtt{L}, \mathtt{F}); \sigma) = \mathtt{NO}}$$

$$\frac{\mathtt{MP}(\mathtt{H}; S; \mathtt{L}; \sigma) = (S', \sigma') \quad \mathtt{MP}(\mathtt{H}; S'; \mathtt{F}; \sigma') = R}{\mathtt{MP}(\mathtt{H}; S; (\mathtt{L}, \mathtt{F}); \sigma) = R}$$

Figure 5. Selected Rules of MP

is sound with regard to the semantics modulo the soundness of axioms.

Lemma 1 (Soundness of Logical Deduction)
If $\Upsilon = \Upsilon_I, \Upsilon_A$ such that Υ_A is sound with regard to Υ_I, and $\Omega \,|\, \Upsilon; \Delta \Longrightarrow \mathtt{F}$, and σ is a grounding substitution for Ω, and $\mathtt{H} \vDash^{\Upsilon_I} \sigma(\Delta)$, then $\mathtt{H} \vDash^{\Upsilon_I} \mathtt{F}$.

3.4 Pattern-Matching Algorithm

The pattern-matching algorithm (MP) determines if a given program heap satisfies a formula. We implements 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 order. In evaluating a clause body, we evaluate the formulas in the body in left-to-right order as they appear.

Figure 5 is a list of selected rules of MP; the complete set of rules is defined in Appendix A. MP takes four arguments: the heap $\mathtt{H}$, a set of locations S that are not usable; a formula $\mathtt{F}$, which may contain uninstantiated variables, and a substitution σ for free variables in $\mathtt{F}$. It either succeeds and returns a substitution for all the free variables in $\mathtt{F}$, and the locations used in proving $\mathtt{F}$, or fails and returns NO. The set of locations is used to deal with 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 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 state predicates, and for clauses whose body contains user-defined predicates, there has to be at least one state predicate that precedes the first user-defined predicate in the body. The above restrictions are also sufficient for the inductive definitions to generate well-founded semantics. We statically check these restrictions at compile time.

3.4.2 Mode Analysis

The mode analysis is also a compile time static check. In the mode analysis, we use context Π to keep track of both the groundness and the safety properties of arguments. The definition of Π is given below. We use `var` to range over stack variables $\$x$ and variables

x. Π contains ground integer arguments and pairs of ground pointer terms and their safety properties.

$$\textit{Ground Ctx} \quad \Pi \quad ::= \quad \cdot \mid \Pi, \mathtt{var} \mid \Pi, (\mathtt{var}, \mathtt{s}) \mid \Pi, (n, \mathtt{yes})$$

We define $\Pi(\mathtt{tm})$ to be the safety property associated with tm in Π. If tm is an integer and not in the domain of Π, then $\Pi(\mathtt{tm})$ is no. We use notation $\Pi\bar{\cup}\{(\mathtt{tm}, \mathtt{s})\}$ to represent a context that is the same as Π except that the safety property associated with tm is the stronger of s and $\Pi(\mathtt{tm})$ (yes is stronger than no). If tm is not in Π, then it is a simple union operation. We use $\mathtt{s}_1 \le \mathtt{s}_2$ to denote that $\mathtt{s}_1$ is stronger or equal to $\mathtt{s}_2$.

The type and mode judgment for formulas has the form $\Xi; \Omega; \Pi \vdash \mathtt{F} : (\mathtt{pt}, \Pi')$. Context Ξ maps predicate names to their types. Context Ω maps free variables in the judgment to their types. All the terms in Π are ground before proving F. Furthermore, if the pair $(\mathtt{tm}, \mathtt{yes})$ is in Π, then tm is a valid pointer on the heap. pt is the type of F. Context Π' keeps track of the groundness and safety properties of arguments after the execution of F. During mode analysis, we also check that the predicates are supplied with arguments of the right types, consequently, the mode analysis uses the typing judgments for expressions ($\Omega \vdash_e e : \mathbf{t}$) to type-check arguments. The typing rules for expressions are standard, and can be found in Appendix F. In addition, for the unification to be well-defined, a term at an output position must be either an integer or a variable. The special judgment $\Omega \vdash_v e : \mathbf{t}$ is the same as the that for expressions, except that it restricts e to be either a number or a variable.

The complete list of mode checking rules is in our Appendix B. Here we explain a few key rules.

Built-in Predicates Now we explain the mode analysis for the built-in predicate equality and struct.

We listed the rules for equality below The first argument is always an output and the second argument is an input. The first rule is when both of the arguments are integers. Since the second argument is an input, we check that all the free variables in $\mathtt{tm}_2$ are in the Π context ($\mathtt{tm}_2$ is ground before the evaluation). The Π' context contains the free variable of $\mathtt{tm}_1$, since $\mathtt{tm}_1$ is unified with $\mathtt{tm}_2$ and is ground after the execution.

The last two rules are when both of the arguments have pointer types. There are two cases. The first one is when $\mathtt{tm}_1$ is not in Π context. We check that $\mathtt{tm}_2$ is ground, and we add $\mathtt{tm}_1$ to have the same safety property as $\mathtt{tm}_2$ into the Π' context. The second case is when $\mathtt{tm}_1$ is in the Π context: then the safety property of $\mathtt{tm}_1$ and $\mathtt{tm}_2$ should both be updated to the stronger of the two.

$$\frac{\Omega \vdash_v \mathtt{tm}_1 : \mathbf{int} \quad \Omega \vdash_e \mathtt{tm}_2 : \mathbf{int} \quad \mathtt{FV}(\mathtt{tm}_2) \subset \mathit{dom}(\Pi)}{\Xi; \Omega; \Pi \vdash \mathtt{tm}_1 = \mathtt{tm}_2 : (\mathtt{o}, \Pi \cup \{\mathtt{FV}(\mathtt{tm}_1)\})}$$

$$\frac{\begin{array}{c}\Omega \vdash_v \mathtt{tm}_1 : \mathbf{ptr}(\mathbf{P}) \qquad \Omega \vdash_e \mathtt{tm}_2 : \mathbf{ptr}(\mathbf{P}) \\ \Pi(\mathtt{tm}_2) = \mathtt{s} \qquad \mathtt{tm}_1 \notin \mathit{dom}(\Pi)\end{array}}{\Xi; \Omega; \Pi \vdash \mathtt{tm}_1 = \mathtt{tm}_2 : (\mathtt{o}, \Pi \cup \{(\mathtt{tm}_1, \mathtt{s})\})}$$

$$\frac{\begin{array}{c}\Omega \vdash_v \mathtt{tm}_1 : \mathbf{ptr}(\mathbf{P}) \qquad \Omega \vdash_e \mathtt{tm}_2 : \mathbf{ptr}(\mathbf{P}) \\ \Pi(\mathtt{tm}_1) = \mathtt{s}_1 \qquad \Pi(\mathtt{tm}_2) = \mathtt{s}_2 \qquad \mathtt{s} = \mathtt{max}(\mathtt{s}_1, \mathtt{s}_2)\end{array}}{\Xi; \Omega; \Pi \vdash \mathtt{tm}_1 = \mathtt{tm}_2 : (\mathtt{o}, \Pi\bar{\cup}\{(\mathtt{tm}_1, \mathtt{s})\}\bar{\cup}\{(\mathtt{tm}_2, \mathtt{s})\})}$$

The modes for struct is declared by the programmer, since the number of fields is defined by the programmer. However, we do check that the mode for the first argument is (+,yes,yes), which specifies that the starting address has to be safe. The strongest form of mode declaration for struct is one in which all the arguments but the first one are output and the pointer arguments are safe. One example is the mode declaration of struct in Figure 2. Such strong declaration is safe because the arguments are read from a heap that has a "closed shape". Recall that pointers read from the heap of closed shapes are valid. This strong form is crucial in making programs pass static checks for memory safety. For example, a typical pattern `[root r, struct r (d, next), list next]` would require next to be a valid pointer after proving `struct r (d, next)` and before proving `list next`. However, we do not restrict the mode declaration of struct to take the strongest form. As long as the program passes the static checks, all is happy.

In the mode-checking rule for struct below, we check that the input arguments are in Π, and the safe pointer arguments are associated with the yes property in Π. Π' is the union of Π and the output integer arguments, and the pair of argument and the specified safety property for output pointer arguments. For the argument of mode $(+, \mathtt{no}, \mathtt{yes})$, Π' should update its safety property to be safe.

$$\frac{\begin{array}{l}\Xi(\mathtt{struct}) = ((+, \mathtt{yes}, \mathtt{yes})\, \mathbf{ptr}(\mathbf{P})) \to (\mathtt{m}_1\, \mathbf{t}_1) \cdots \to (\mathtt{m}_n\, \mathbf{t}_n) \to \mathtt{o} \\ \forall i \in [1, n], \\ \left\{\begin{array}{ll}\Omega \vdash_e \mathtt{tm}_i : \mathbf{int}, \mathtt{FV}(\mathtt{tm}_i) \in \Pi & \mathtt{m}_i\, \mathbf{t}_i = +\, \mathbf{int} \\ \Omega \vdash_e \mathtt{tm}_i : \mathbf{ptr}(\mathbf{P}), \Pi(\mathtt{tm}_i) \le \mathtt{s}_1 & \mathtt{m}_i\, \mathbf{t}_i = (+, \mathtt{s}_1, s_2)\, \mathbf{ptr}(\mathbf{P}) \\ \Omega \vdash_v \mathtt{tm}_i : \mathbf{t}_i & \mathtt{m}_i = - \text{ or } (-, \mathtt{s}_1, s_2)\end{array}\right. \\ \quad \Pi' \quad = \quad \Pi \cup \{\mathtt{FV}(\mathtt{tm}_j) \mid m_j t_j = -\, \mathbf{int}\} \\ \qquad\qquad\quad \bar{\cup}\{(\mathtt{tm}_k, \mathtt{s}_2) \mid m_k = (-, \mathtt{s}_1, \mathtt{s}_2)\} \\ \qquad\qquad\quad \bar{\cup}\{(\mathtt{tm}_i, \mathtt{yes}) \mid m_i = (+, \mathtt{no}, \mathtt{yes})\}\end{array}}{\Xi; \Omega; \Pi \vdash \mathtt{struct}\, \mathtt{tm}\, \mathtt{tm}_1 \cdots \mathtt{tm}_n : (\mathtt{o}, \Pi')}$$

User-defined Predicates The rules for checking user-defined predicate are straightforward. We check that Π contains all the input arguments and associates pointer arguments with the right safety property. The Π' context is produced in the same way as the rule for struct. Note that this rule is used under the assumption that the declared modes for user-defined predicates are correct. The real check of the correctness of these mode declarations is done when we mode-check the clauses, which we will explain shortly.

Conjunction Since the execution order is from left to right, we check the second formula in a multiplicative conjunction with the Π' generated by checking the first one.

$$\frac{\Xi; \Omega; \Pi \vdash \mathtt{L} : (\mathtt{o}, \Pi') \quad \Xi; \Omega; \Pi' \vdash \mathtt{F} : (\mathtt{o}, \Pi'')}{\Xi; \Omega; ; \Pi \vdash \mathtt{L}, \mathtt{F} : (\mathtt{o}, \Pi'')}$$

Clause As we mentioned earlier, to check the correctness of the mode declarations for user-defined predicates, we need to mode-check the inductive definitions, the rule for which is as follows.

$$\frac{\begin{array}{c}\Xi(\mathbf{P}) = \mathtt{pt} \quad \Omega = \mathtt{infer}(\mathbf{P}\, \overline{x}) \\ \Pi = \{x_j | \mathtt{pt}_j = +\, \mathbf{int}\} \cup \{(x_j, \mathtt{s}_{in}) | \mathtt{pt}_j = (+, \mathtt{s}_{in}, \mathtt{s}_{out})\, \mathbf{ptr}(\mathbf{P})\} \\ \Xi; \Omega; \Pi \vdash \mathbf{P}\, \overline{x} : (\mathtt{o}, \Pi') \\ \forall i \in [1, n], \quad \Omega' = \mathtt{infer}(\mathtt{F}_i) \\ \Xi; \Omega', \Omega; \Pi \vdash \mathtt{F}_i : (\mathtt{o}, \Pi'') \qquad \Pi'' < \Pi'\end{array}}{\Xi \vdash ((\mathtt{F}_1; \cdots; \mathtt{F}_n) \multimap \mathbf{P}\, \overline{x})\ \mathsf{OK}}$$

We use $\mathtt{pt}_i$ to denote the type of the i^{th} curried argument in pt. The arguments' types are easily inferred from the type declaration of the predicates. We use $\mathtt{infer}(\mathtt{F})$ to denote the inferred type bindings of the free variables in F. The Π and Π' context come directly from the mode declaration of $\mathbf{P}$, and they contain the ground integer arguments and the pairs of ground pointer argument and its declared safety property before and after the evaluation of $\mathbf{P}$ respectively. When predicate $\mathbf{P}$ is executed, the bodies of $\mathbf{P}$ are executed, and the declared output arguments of $\mathbf{P}$ have to be provided by the execution of its bodies. Therefore, we need to check that Π'', which contains the groundness and safety properties obtained from the evaluation of the body of the clause, is compatible with Π, which contains the declared modes after the evaluation of the predicate $\mathbf{P}$. We use a subtyping relation ($\Pi'' < \Pi'$) to describe the compatibility between the two contexts. The subtyping relationship means that all the ground terms in Π', have to be in Π'', and all the safe pointers in Π' have to be safe in Π''.

3.4.3 Formal Results

In this section we present the formal properties of MP.

We proved the following theorem that states that MP terminates if the inductive definitions are well-formed. Judgment $\vdash \mathtt{I}\ \mathtt{well}-\mathtt{formed}$ checks the termination constraints.

Theorem 2 (Termination of MP)
If for all $\mathtt{I} \in \Upsilon$, $\vdash \mathtt{I}\ \mathtt{well}-\mathtt{formed}$*, then* $\mathtt{MP}(\Upsilon)$ *always terminates.*

We also proved the correctness of MP. Notice that MP only implements partial backtracking; we try all the clause bodies for one predicate, but we do not backtrack for a second solution. We can afford to do so because we take advantage of the *Uniqueness of Shape Matching* requirement, which requires that given a heap, the heap described by the predicate of top-level shape name is unique (Section 2.6). This requirement yields the result that the unification for uninstantiated existential variables for any predicate that is defined in the shape signature and has ground input arguments, is unique. The following requirement is the generalized formal definition of the one introduced in Section 2.6. Intuitively it means that the input arguments of a predicate determines the shape described by that predicate.

Requirement: Uniqueness of Shape Matching

If $\Xi;\Omega;\Pi \vdash \mathtt{L} : (\mathtt{o},\Pi')$, and $\forall x \in dom(\Pi).\ x \in dom(\sigma)$ and $\mathtt{H}_1 \vDash \sigma_1(\mathtt{L})$, and $\mathtt{H}_2 \vDash \sigma_2(\mathtt{L})$, and $\sigma \subseteq \sigma_1$, and $\sigma \subseteq \sigma_2$, and $\mathtt{H}_1 \subset \mathtt{H}$, and $\mathtt{H}_2 \subset \mathtt{H}$, then $\mathtt{H}_1 = \mathtt{H}_2$, and $\sigma_1 = \sigma_2$.

We proved that MP is complete and correct with regard to the semantics if all the user-defined predicates comply with the uniqueness restriction.

Theorem 3 (Correctness of MP)
If $\Xi;\Omega;\Pi \vdash \mathtt{F} : (\mathtt{o},\Pi')$*, 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* $\mathtt{H}' \vDash \sigma'(\mathtt{F})$*, 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 exists no* $\mathtt{H}'$, $dom(\mathtt{H}') \subset (dom(\mathtt{H}) - S)$*, and* σ', $\sigma \subset \sigma'$*, such that* $\mathtt{H}' \vDash \sigma'(\mathtt{F})$

Finally, we proved that if the heap has a closed shape, the inductive definitions are well-formed, the goal formula is well-moded under Π, all the input arguments in Π are ground, and all the safe pointers in Π are valid pointers on the heap, then MP never dereferences dangling pointers when traversing the heap.

Theorem 4 (Safety of MP)
If for all $\mathtt{I} \in \Upsilon$, $\vdash \mathtt{I}\ \mathtt{well}-\mathtt{formed}$, $\mathbf{P}$ *is a closed shape, and* $\Xi = \Lambda(\mathbf{P})$, $\mathtt{H}_1 \vDash \mathbf{P}(l)$, $\Xi;\Omega;\Pi \vdash \mathtt{F} : (\mathtt{o},\Pi')$*, and* $\forall \mathtt{var} \in dom(\Pi).\ \mathtt{var} \in dom(\sigma)$*, and* $S \subset dom(\mathtt{H}_1)$*, and* $\forall \mathtt{tm}$ *such that* $\Pi(\mathtt{tm}) = \mathtt{yes}$, $\sigma(\mathtt{tm}) \in \bar{dom}(\mathtt{H}_1)$ *or* $\sigma'(\mathtt{tm}) = 0$ *then*

- *either* $\mathtt{MP}(\Upsilon)(\mathtt{H}_1 \uplus \mathtt{H}_2;S;\mathtt{F};\sigma) = (S',\sigma')$*, and* MP *will not access location* l *if* $l \notin dom(\mathtt{H}_1)$*, and* $\forall \mathtt{var} \in dom(\Pi').\ \mathtt{var} \in\_ dom(\sigma')$*, and* $\forall \mathtt{tm}$ *such that* $\Pi'(\mathtt{tm}) = \mathtt{yes}$*, then* $\sigma'(\mathtt{tm}) \in \bar{dom}(\mathtt{H}_1)$ *or* $\sigma'(\mathtt{tm}) = 0$
- *Or* $\mathtt{MP}(\Upsilon)(\mathtt{H}_1 \uplus \mathtt{H}_2;S;\mathtt{F};\sigma) = \mathtt{NO}$*, and* MP *will not access location* l *if* $l \notin dom(\mathtt{H}_1)$*.*

Since the program heap often contains many data structures, Theorem 4 takes the frame property into account: MP is safe on a larger heap. 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 does 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, the pattern-matching algorithm into the operational semantics, and thereby integrate the verification technique into the language. First, we introduce various syntactic constructs in our language, then we define the formal operational semantics, next we explain the type system. Lastly, we show an example of a function that inserts a key into an ordered list.

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

Figure 6. Syntax of Language Constructs

4.1 Syntax

A summary of the syntax of our core language is shown in Figure 6. 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 variables bound 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. Shape formulas Shp are the multiplicative conjunction of a set of predicates, the first of which is the special root predicate indicating the starting address of the shape. A shape pattern pat can be either a query pattern ($?\,[\mathtt{Shp}]$) or a deconstructive pattern ($:[\mathtt{Shp}]$). 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 ($?\,[\mathtt{Shp}]$). These two patterns are treated the same operationally, but differently in the type system. The deconstructive patterns generate formulas describing accessible portions of the heap, but the query patterns don't. The conditional expressions in if statements and while loops are composed of variable bindings and a conjunctive clause cc, which describes the arithmetic constraints and the shape patterns of one or more disjoint data structures. The free variables in cc are bound in the variable bindings. 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 shape variables 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.

	$(E;\mathtt{H};\mathtt{stmt}) \longmapsto (E';\mathtt{H}';\mathtt{stmt}')$
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	$(E;\mathtt{H};\$s\ :=\ \{\overline{x{:}\mathbf{ptr(P)}}\}\ [\mathtt{root}\ (v), \mathtt{F}])$ $\longmapsto (E[\$s := v'];\mathtt{H}';\mathtt{skip})$ where $(\mathtt{H}', v') =$ $\mathtt{CreateShape}(\mathtt{H}, \{\overline{x{:}\mathbf{ptr(P)}}\}\ [\mathtt{root}\ (v), \mathtt{F}], \mathbf{P})$
If-t	$(E;\mathtt{H};\mathtt{if}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2)$ $\longmapsto (E;\mathtt{H};\sigma(\mathtt{stmt}_1))$ if $\llbracket \mathtt{cc} \rrbracket_E = (\mathtt{F}, \sigma')$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma') = (SL;\sigma)$
If-f	$(E;\mathtt{H};\mathtt{if}\ \{\overline{x{:}\mathbf{t}}\}\ cc\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2)$ $\longmapsto (E;\mathtt{H};\mathtt{stmt}_2)$ if $\llbracket cc \rrbracket_E = (\mathtt{F}, \sigma)$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma) = \mathtt{NO}$
while-t	$(E;\mathtt{H};\mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt})$ $\longmapsto (E;\mathtt{H};(\sigma(\mathtt{stmt}_1)\ ;\ \mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt}))$ if $\llbracket \mathtt{cc} \rrbracket_E = (\mathtt{F}, \sigma')$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma') = (SL;\sigma)$
while-f	$(E;\mathtt{H};\mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt}) \longmapsto (E;\mathtt{H};\mathtt{skip})$ if $\llbracket \mathtt{cc} \rrbracket_E = (\mathtt{F}, \sigma)$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma) = \mathtt{NO}$
switch-t	$(E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \{\overline{x{:}\mathbf{t}}\}\ ([\mathtt{root}\ (x_i), \mathtt{F}] \rightarrow \mathtt{stmt}_k)$ $\vert \mathtt{bs}\ ) \longmapsto (E;\mathtt{H};\sigma(\mathtt{stmt}_k))$ if $\mathtt{MP}(\mathtt{H};E(\mathtt{F});\emptyset;\{E(\$s)/x_i\}) = (SL;\sigma)$
switch-f	$(E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \{\overline{x{:}\mathbf{t}}\}\ ([\mathtt{root}\ (x_i), \mathtt{F}] \rightarrow \mathtt{stmt}_k)$ $\vert\ \mathtt{bs}\ ) \longmapsto (E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \mathtt{bs}\ )$ if $\mathtt{MP}(\mathtt{H};E(\mathtt{F});\emptyset;\{E(\$s)/x_i\}) = \mathtt{NO}$
fail	$(E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \{\overline{x{:}\mathbf{t}}\}\ ([\mathtt{root}\ (x_i), \mathtt{F}] \rightarrow \mathtt{stmt}_k))$ $\longmapsto (E;\mathtt{H};\mathtt{fail})$ if $\mathtt{MP}(\mathtt{H};E(\mathtt{F});\emptyset;\{E(\$s)/x_i\}) = \mathtt{NO}$

Figure 7. Selected Operational Semantics

4.2 Operational Semantics

In this section, we will formally 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 complete set of rules or operational semantics, and other auxiliary definitions can be found in a summary in Appendix E.

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 7 is a list of selected rules.

We write $E(\$x)$ and $E(\$s)$ to denote the value E maps $\$x$ and $\$s$ to, and we also call this value the run-time value of these variables. We write $E(F)$ to denote the formula with run-time values substituted for the stack 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`.

To create a data structure, programmers use the shape assignment statement. During the evaluation of this statement, the heap is updated according to the shape formulas in the assignment statement. In the end, the root of the new shape is stored in the shape variable $\$s$. The core procedure is `CreateShape`, which takes the current heap, the shape formula, and the shape name $\mathbf{P}$ as arguments, and returns the updated heap and the root of the new shape. To explain how `CreateShape` works, we first define a few macros. We use $\mathtt{size}(\mathbf{P}, \mathtt{struct})$ to denote the size of the each for a heap of shape $\mathbf{P}$. The size is decided by the type declaration of `struct` in $\mathbf{P}$'s signature. For example, for `listshape` in Figure 2, `size(listshape, struct) = 2`. We define the macro `alloc(H, k)` to allocate a tuple of size (k+1) on `H`, store `k` in the first location, initialize all other fields to be 0, and return the new heap and the starting address of the tuple. The definition of the `CreateShape` procedure is given below.

```
$\mathtt{CreateShape}(\mathtt{H}, \{\overline{x{:}\mathbf{ptr(P)}}\}(\mathtt{root}\ n, \mathtt{F}), \mathbf{P}) = (\mathtt{H}', v')$
    1. $k = \mathtt{size}(\mathbf{P}, \mathtt{struct})$
    2. $(\mathtt{H}_1, l_1) = \mathtt{alloc}(\mathtt{H}, k), \cdots, (\mathtt{H}_n, l_n) = \mathtt{alloc}(\mathtt{H}_{n-1}, k)$
    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 $(\mathtt{struct}\ v\ v_1 \cdots v_k) \in \mathtt{F}'$
    5. return $(\mathtt{H}', n\,[\,\overline{l}\,/\,\overline{x}\,])$
```

In this procedure, we allocate a tuple on the heap for each bound variable. The addresses returned by `alloc` are bound to the variables as their run-time values. Then, we modify the heap according to the shape formula $\mathtt{F}'$, which results from substituting the run-time values for the bound variables in `F`. For each $\mathtt{struct}\ \mathtt{v}_0\ \overline{\mathtt{v}}$ in $\mathtt{F}'$, the heap cell at location $\mathtt{v}_0 + \mathtt{i}$ is updated to store value $\mathtt{v}_\mathtt{i}$.

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. If `MP` returns `NO`, the false branch is evaluated. Notice that the conditional expression is not in the form of a logical formula; therefore, we need to convert the conditional expression to its equivalent formula $\mathtt{F}_{cc}$, before invoke the pattern-matching procedure `MP`. We define $\llbracket\, cc \,\rrbracket_E$ 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 $E(\$s) = 100$, then

$$\llbracket\, \$\mathtt{s}\ :\ [\mathtt{root}\ \mathtt{r},\ \mathtt{struct}\ \mathtt{r}\ (\mathtt{d},\ \mathtt{next}),\ \mathtt{list}\ \mathtt{next}] \,\rrbracket_\mathtt{E} \\ = ((\mathtt{struct}\ \mathtt{r}\ (\mathtt{d},\ \mathtt{next}),\ \mathtt{list}\ \mathtt{next}), \{100/\mathtt{r}\}).$$

The substitution $\sigma = \{100/r\}$ comes from the fact that the starting address of this shape is `r`, and at run time, this address is stored in $\$s$. Therefore, `r` should be unified with $E(\$s)$, which is 100. `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 $\llbracket\, \mathtt{cc} \,\rrbracket_E$.

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.

The shape patterns in switch statements are special cases of conditional expressions in that the shape variable being branched on is the only shape being considered. `MP` is called the same way. If the current shape pattern succeeds, the branch body is evaluated; otherwise the next branch is considered. If the pattern of the last branch failed, then the run-time statement `fail` is evaluated.

4.3 Type System

As we briefly mentioned in Section 2, 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 are the valid shapes, and what is the structure of the accessible heap.

Below are the contexts used in the typing judgments:

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. The difference between Γ and Θ is that Γ contains the initialized shape variable, while Θ contains the uninitialized shape variables. Context Δ is a set of formulas that describe the accessible portions of the heap. Context Γ and Δ describe the entire program heap. The intuitive meaning of Γ is that each shape variable $\$s$ in Γ holds the starting address of a distinct piece of heap that has the shape $\Gamma(\$s)$. For example, if $\Gamma = \$s{:}\mathtt{listshape}$, Δ = `struct 400 (11, 0)`, and the environment is $E = \$s \mapsto 100$, then the current heap must satisfy formula (`listshape 100, struct 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	$\Omega; \Gamma; \Pi \vdash \mathtt{cc} : \Pi'$
Statement typing	$\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} : (\Gamma'; \Theta'; \Delta')$

For simplicity, we omit the contexts for shape signatures Λ, axioms Υ, and function type bindings Φ, from the above judgments. A complete list of the typing rules is listed in Appendix F. Here we focus on explaining the statement typing rules.

Conjunctive Clause Typing The typing judgment of 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.

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

Here, shape variable $\$s$ is deconstructed by shape pattern $(\mathtt{root}\, y, \mathtt{F}_A, \mathtt{F})$, where $\mathtt{F}_A$ are the arithmetic formulas. If the 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 makes sure 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 $\$s$. Since the heaplet H$'$ that $\$s$ points to before the pattern matching also has shape $\mathbf{P}$ v, by the uniqueness of shapes we know that H is exactly the same as H$'$.

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

$$\frac{\Gamma(\$s) = \mathbf{P} \quad \Lambda(\mathbf{P}) = \Xi \quad \Xi; \Omega; \Pi \bar{\cup} \{x{:}\mathtt{yes}\} \vdash \mathtt{F} : (\mathtt{o}, \Pi')}{\Omega; \Gamma; \Pi \vdash \$s : [\mathtt{root}\, x, \mathtt{F}] : \Pi'}$$

Since $\$s$ points to a valid shape on the heap, its root pointer is an valid pointer on the current heap. 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 8.

The rule for if statements first collects the typing information of the bound variables into a new context Ω'. We assume alpha-renaming is applied whenever necessary. 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 Π context, which contain groundness and safety properties of the arguments when cc is evaluated, depends on the variable context Ω, and is denoted by $\mathtt{ground}(\Omega)$. 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.

$$\begin{aligned} \mathtt{ground}(\Omega) = \quad & \{\mathtt{var} \mid \Omega(\mathtt{var}) = \mathbf{int}\} \\ & \cup \{(\mathtt{var}, \mathtt{no}) \mid \Omega(\mathtt{var}) = \mathbf{ptr}(\mathbf{P})\} \end{aligned}$$

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 $\$s$ to be assigned to is uninitialized to prevent memory leaks. All the shape variables in Γ point to a valid shape in the heap, so assigning $\$s$ to point to another shape makes us lose the pointer to the shape $\$s$ originally points to. The logical judgment checks that the shape formula entails the shape we want. The rest of the judgments means 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 locations allocated. It looks complicated because we allow updates to the heap cells during construction. For example, if capability `struct l (5,0)` is available, then we allow `struct 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 switch statement requires that each branch results in the same program state. Each branch can be viewed as a simplified if statement with only a true branch, and the conditional expression only considers one shape pattern.

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 A More Complicated Example

Now we will demonstrate the expressiveness of our language through an insert function in Figure 9 which inserts an integer into a list and maintains the ascending order of its keys.

Function insert is an implementation of an algorithm that uses pointer $p to traverse the list until it reaches the end of the list or the data field under $p is greater than or equal to the key to be inserted. A second pointer $pre points to the parent of $p. The new node should be inserted between $pre and $p. The first argument of insert, $s, has listshape type and holds the starting address of the ascending list. The second argument, $k, is the integer to be inserted. The local stack variables are the two traversing pointers, `$p and $pre`. The if statement between line 4 and 6 initializes both $p and $pre to point to the head of the list. The while loop from line 7 through 11 traverses the list. The conditional expression between keywords while and do examines the heaplet pointed to by $\$s$ to see if between the head of the list and the traversing pointer $p is a list segment, and $p points to a pair of values key and next such that next points to a list and key is less than $k. The body of the while loop advances the traversing pointers. The switch statement between line 12 and 20 inserts the key into the list. After the while loop, there are two possibilities. One is that the key should be inserted before the head of the list and $pre and $p both point to the head of the list. The other is that $pre is the parent of $p, and key should be inserted between these two pointers. The switch statement branches on these two cases and update the heap appropriately.

$$\dfrac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad dom(\Omega') \subset \texttt{FV}(\texttt{cc}) \quad \Omega', \Omega; \Gamma \vdash \texttt{cc} \;:\; (\Gamma', \Theta', \Delta') \\ \Omega', \Omega; \Gamma'; \Theta, \Theta'; \Delta, \Delta' \vdash \texttt{stmt}_1 \;:\; (\Gamma''; \Theta''; \Delta'') \\ \Omega; \Gamma; \Theta; \Delta \vdash \texttt{stmt}_2 \;:\; (\Gamma''; \Theta''; \Delta'') \\ \Pi = \texttt{ground}(\Omega) \qquad \Omega', \Omega; \Gamma; \Pi \vdash \texttt{cc} \;:\; \Pi' \\ \forall x_i \in dom(\Omega'), x_i \in dom(\Pi') \end{array}}{\begin{array}{r}\Omega; \Gamma; \Theta; \Delta \vdash \texttt{if } \{\overline{x{:}\mathbf{t}}\} \texttt{ cc then stmt}_1 \texttt{ else stmt}_2 \\ :\; (\Gamma''; \Theta''; \Delta'')\end{array}} \; \textit{if}$$

$$\dfrac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad dom(\Omega') \subset \texttt{FSV}(\texttt{cc}) \\ \Omega', \Omega; \Gamma \vdash \texttt{cc} \;:\; (\Gamma', \Theta', \Delta') \\ \Omega', \Omega; \Gamma'; \Theta, \Theta'; \Delta, \Delta' \vdash \texttt{stmt} \;:\; (\Gamma; \Theta; \Delta) \\ \Pi = \texttt{ground}(\Omega) \qquad \Omega', \Omega; \Gamma; \Pi \vdash \texttt{cc} \;:\; \Pi' \\ \forall x_i \in dom(\Omega'), x_i \in dom(\Pi') \end{array}}{\Omega; \Gamma; \Theta; \Delta \vdash \texttt{while } \{\overline{x{:}\mathbf{t}}\} \texttt{ cc do stmt} \;:\; (\Gamma; \Theta; \Delta)} \; \textit{while}$$

$$\dfrac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{ptr}(\mathbf{P})} \qquad \Theta = \Theta', (\$s : \mathbf{P}) \\ \Omega', \Omega \,|\, \Upsilon; \texttt{F} \Longrightarrow \mathbf{P}(v) \\ \Delta_x = \{\texttt{struct } x_i\, \overline{e} \,|\, \texttt{struct } x_i\, \overline{e} \in \texttt{F}\} \\ \Delta = \Delta', \Delta'' \quad \texttt{F} = \Delta_x, \Delta_F \quad \forall \texttt{Pu}, \texttt{Pu} \in \Delta'' \text{ iff } \texttt{Pu} \in \Delta_F \\ \forall \texttt{Ps} = \texttt{struct tm}\, \overline{e}, \texttt{Ps} \in \Delta'' \text{ iff } \texttt{struct tm}\, \overline{e}' \in \Delta_F \end{array}}{\begin{array}{r}\Omega; \Gamma; \Theta; \Delta \vdash \$s \;:=\; \{\overline{x{:}\mathbf{ptr}(\mathbf{P})}\}[\texttt{root}\,(v), \texttt{F}] \\ :\; (\Gamma', (\$s{:}\mathbf{P}); \Theta'; \Delta')\end{array}} \; \textit{assign-shape}$$

$$\dfrac{\text{For all i, } 1 \leq i \leq n, \quad \Omega; \Gamma; \Theta; \Delta \vdash_{\$s} \texttt{b}_{\,i} \;:\; (\Gamma'; \Theta'; \Delta')}{\Omega; \Gamma; \Theta; \Delta \vdash \texttt{switch } \$s \texttt{ of bs} \;:\; (\Gamma'; \Theta'; \Delta')} \; \textit{switch}$$

$$\dfrac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad \Gamma(\$s) = \mathbf{P} \qquad \Xi = \Lambda(\mathbf{P}) \\ \Xi; \Omega', \Omega; \Gamma; \texttt{ground}(\Omega) \bar{\cup} \{x_i{:}\texttt{yes}\} \vdash \texttt{F} \;:\; \Pi \\ \forall x_i \in dom(\Omega'), x_i \in dom(\Pi) \\ \Omega', \Omega \,|\, \Upsilon; \texttt{F} \Longrightarrow \mathbf{P}(x_i) \\ \Omega', \Omega; \Gamma; \Theta; \Delta \vdash \texttt{stmt} \;:\; (\Gamma'; \Theta'; \Delta') \end{array}}{\Omega; \Gamma; \Theta; \Delta \vdash_{\$s} \{\overline{x{:}\mathbf{t}}\}\; ?[\texttt{root}\,(x_i), \texttt{F}] \to \texttt{stmt} \;:\; (\Gamma'; \Theta'; \Delta')} \; \textit{pat-?}$$

$$\dfrac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad \Gamma = \Gamma', \$s{:}\mathbf{P} \\ \Xi = \Lambda(\mathbf{P}) \qquad \texttt{Shp} = \texttt{root}\,(x_i), \texttt{F}_A, \texttt{F} \\ \Xi; \Omega; \Omega'; \Gamma; \texttt{ground}(\Omega) \bar{\cup} \{x_i{:}\texttt{yes}\} \vdash \texttt{F} \;:\; \Pi \\ \forall x_i \in dom(\Omega'), x_i \in dom(\Pi) \\ \Omega', \Omega \,|\, \Upsilon; \texttt{F}_A, \texttt{F} \Longrightarrow \mathbf{P}(x_i) \qquad \texttt{FV}(\texttt{F}) \cap \Omega_{\$} = \emptyset \\ \Omega', \Omega; \Gamma'; \Theta, \$s{:}\mathbf{P}; \Delta, \texttt{F} \vdash \texttt{stmt} \;:\; (\Gamma''; \Theta'; \Delta') \end{array}}{\Omega; \Gamma; \Theta; \Delta \vdash_{\$s} \{\overline{x{:}\mathbf{t}}\}\; {:}[\texttt{Shp}] \to \texttt{stmt} \;:\; (\Gamma''; \Theta'; \Delta')} \; \textit{pat-:}$$

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

Figure 8. Selected Statement Typing Rules

```
listshape insert(listshape $s, int $k){

  ptr(listshape) $pre := 0;
  ptr(listshape) $p := 0;

  if $s?[root x, list x]
  then {$pre := x; $p := x};
  else skip

  while ($s?[root x, listseg x $p,
             struct $p (key, next), list next,
             $k > key ])
  do {$pre := $p; $p := next};

  switch $s of [root x, list x,
                ($pre = x), ($pre = $p)]
      -> $s := [root n, struct n ($k, x), list x]

   |  [root x, y = $pre, listseg x y,
       struct y (key, next), list next]
      -> $s : = [root x, listseg x y,
                  struct y (key, n),
                  struct n  ($k, next),
                  list next]
}
```

Figure 9. list_insert

4.5 Progress and Preservation

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 $\vdash (Es; \texttt{H}; S; \texttt{fb})$ OK. We proved the following progress and preservation theorem for our language.

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

5. 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 a notation based on context-free 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 [22], 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 [21, 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, 15] and automated theorem provers [2, 16], 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 [18], the basis for separation logic coincide exactly on this fragment). We call our logic "linear" since it's 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 [25, 3, 23, 27, 8, 17, 28].

6. Current and Future Work

We are currently working on an implementation of our system, and up to now, the prototype works for singly linked lists.

One important piece of future work is to identify the necessary conditions for programmer-defined inductive definitions to generate a decidable proof system for compile-time checking. In the prototype implementation, we use Lolli [10] to check the proofs. Because the algorithm deployed by Lolli is a naive bottom-up proof search, the type checker could run into infinite loop if the shape formulas are not written carefully. For specific shapes like list and trees, we have checked termination results by hand, just as is done in separation logic. However, we are looking for a more general theorem concerning termination of linear logic programs without function symbols.

For the memory safety of the run-time system, we impose certain well-formedness requirements on the shape signatures. At this point, programmers have to check these requirements by hand, but we believe that these requirements can be checked automatically, or at least semi-automatically.

In the longer term, we would like to develop a whole-program verification process by integrating the research presented here with concurrent work on the development of separation logic.

7. Conclusion

We 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 pointer. Lastly, we integrated all these new ideas into an imperative programming language, for which we are developing 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*, Berlin, June 2002. ACM Press.

[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] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

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

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

[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] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–280, 2004.

[21] 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.

[22] 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.

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

[24] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, MD, 1988.

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

[26] 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.

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

[28] 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.

A. Pattern Matching Algorithm

$$\frac{\texttt{FV}(e_1) \subset dom(\sigma) \quad \texttt{FV}(e_2) \subset dom(\sigma) \quad \sigma(e_1) = \sigma(e_2)}{\texttt{MP}(\texttt{H}; S, e_1 = e_2, \sigma) = (S, \sigma)}$$

$$\frac{\texttt{FV}(e_1) \subset dom(\sigma) \quad \texttt{FV}(e_2) \subset dom(\sigma) \quad \sigma(e_1) \neq \sigma(e_2)}{\texttt{MP}(\texttt{H}; S, e_1 = e_2, \sigma) = \texttt{NO}}$$

$$\frac{x \notin dom(\sigma) \quad \texttt{FV}(e_2) \subset dom(\sigma) \quad v = \sigma(e_2)}{\texttt{MP}(\texttt{H}; S, x = e_2, \sigma) = (S, \sigma \cup \{v/x\})}$$

$$\frac{\texttt{FV}(e_1) \subset dom(\sigma) \quad \texttt{FV}(e_2) \subset dom(\sigma) \quad \sigma(e_1) > \sigma(e_2)}{\texttt{MP}(\texttt{H}; S, e_1 > e_2, \sigma) = (S, \sigma)}$$

$$\frac{\texttt{FV}(e_1) \subset dom(\sigma) \quad \texttt{FV}(e_2) \subset dom(\sigma) \quad \sigma(e_1) \leq \sigma(e_2)}{\texttt{MP}(\texttt{H}; S, e_1 > e_2, \sigma) = (S, \sigma)}$$

$$\frac{\texttt{MP}(\texttt{H}; S, A, \sigma) = \texttt{NO}}{\texttt{MP}(\texttt{H}; S,\ \texttt{not}\ A, \sigma) = (S, \sigma)} \qquad \frac{\texttt{MP}(\texttt{H}; S, A, \sigma) = (S, \sigma)}{\texttt{MP}(\texttt{H}; S,\ \texttt{not}\ A, \sigma) = \texttt{NO}}$$

$$\frac{\sigma(\texttt{tm}) \in S \text{ or } \sigma(\texttt{tm}) = 0}{\texttt{MP}(\texttt{H}; S, \texttt{struct tm tm}_1 \cdots \texttt{tm}_n), \sigma) = \texttt{NO}}$$

$$\frac{\begin{array}{c}\sigma(\texttt{tm}) \neq 0 \qquad \sigma(\texttt{tm}) \notin S \qquad \bar{\texttt{H}}(\sigma(\texttt{tm})) = (v_1, \cdots, v_n) \\ \texttt{MP}(\texttt{H}; S; \texttt{tm}_1 = v_1; \sigma) = (S, \sigma_1) \\ \cdots \\ \texttt{MP}(\texttt{H}; S; \texttt{tm}_k = v_k; \sigma_{k-1}) = \texttt{NO}\end{array}}{\texttt{MP}(\texttt{H}; S; \texttt{struct tm tm}_1 \cdots \texttt{tm}_n; \sigma) = \texttt{NO}}$$

$$\frac{\begin{array}{c}\sigma(\texttt{tm}) \neq 0 \text{ and } \sigma(\texttt{tm}) \notin S \qquad \bar{\texttt{H}}(\sigma(\texttt{tm})) = (v_1, \cdots, v_n) \\ \texttt{MP}(\texttt{H}; S; \texttt{tm}_1 = v_1; \sigma) = (S, \sigma_1) \\ \cdots \\ \texttt{MP}(\texttt{H}; S; \texttt{tm}_n = v_n; \sigma_{n-1}) = (S, \sigma_n)\end{array}}{\texttt{MP}(\texttt{H}; S; \texttt{struct tm tm}_1 \cdots \texttt{tm}_n; \sigma) = (S \cup \{\sigma(x), \cdots, \sigma(x) + n\}, \sigma_n)}$$

$$\frac{\Upsilon(\mathbf{P}) = (\overline{\texttt{F}} \multimap \mathbf{P}\ \overline{y}) \quad \forall i \in [1, k],\ \texttt{MP}(\texttt{H}; S; \texttt{F}_i\,[\,\overline{\texttt{tm}}\,/\,\overline{y}\,]; \sigma) = \texttt{NO}}{\texttt{MP}(\texttt{H}; S; \mathbf{P}\ \overline{\texttt{tm}}; \sigma) = \texttt{NO}}$$

$$\frac{\Upsilon(\mathbf{P}) = (\overline{\texttt{F}} \multimap \mathbf{P}\ \overline{y}) \quad \exists i \in [1, k],\ \texttt{MP}(\texttt{H}; S; \texttt{F}_i\,[\,\overline{\texttt{tm}}\,/\,\overline{y}\,]; \sigma) = (S_i, \sigma_i)}{\texttt{MP}(\texttt{H}; S; \mathbf{P}\ \overline{\texttt{tm}}; \sigma) = (S_i, \sigma_i)}$$

$$\frac{\texttt{MP}(\texttt{H}; S; \texttt{L}; \sigma) = \texttt{NO}}{\texttt{MP}(\texttt{H}; S; (\texttt{L}, \texttt{F}); \sigma) = \texttt{NO}}$$

$$\frac{\texttt{MP}(\texttt{H}; S; \texttt{L}; \sigma) = (S', \sigma') \quad \texttt{MP}(\texttt{H}; S'; \texttt{F}; \sigma') = R}{\texttt{MP}(\texttt{H}; S; (\texttt{L}, \texttt{F}); \sigma) = R}$$

B. Mode Analysis and Related Definitions

$$\boxed{\Pi \bar{\cup} \{(\texttt{tm}, \texttt{s})\} = \Pi'}$$

$$\begin{array}{c}(\Pi, (\texttt{var}, \texttt{no}))\bar{\cup}\{(\texttt{var}, \texttt{s})\} = \Pi, (\texttt{var}, \texttt{s}) \\ \Pi\bar{\cup}\{(n, \texttt{no})\} = \Pi \\ (\Pi, (\texttt{tm}, \texttt{yes}))\bar{\cup}\{(\texttt{tm}, \texttt{s})\} = \Pi, (\texttt{tm}, \texttt{yes}) \\ \Pi\bar{\cup}\{(\texttt{tm}, \texttt{s})\} = \Pi, (\texttt{tm}, \texttt{s}) \text{ if } \texttt{tm} \notin dom(\Pi)\end{array}$$

$$\boxed{\Xi; \Omega; \Pi \vdash \texttt{F} \,:\, (\texttt{pt}, \Pi')}$$

$$\frac{\Omega \vdash_v \texttt{tm}_1 \,:\, \mathbf{int} \quad \Omega \vdash_e \texttt{tm}_2 \,:\, \mathbf{int} \quad \texttt{FV}(\texttt{tm}_2) \subset dom(\Pi)}{\Xi; \Omega; \Pi \vdash \texttt{tm}_1 = \texttt{tm}_2 \,:\, (\texttt{o}, \Pi \cup \{\texttt{FV}(\texttt{tm}_1)\})}$$

$$\frac{\begin{array}{c}\Omega \vdash_v \texttt{tm}_1 \,:\, \mathbf{ptr}(\mathbf{P}) \qquad \Omega \vdash_e \texttt{tm}_2 \,:\, \mathbf{ptr}(\mathbf{P}) \\ \Pi(\texttt{tm}_2) = \texttt{s} \qquad \texttt{tm}_1 \notin dom(\Pi)\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{tm}_1 = \texttt{tm}_2 \,:\, (\texttt{o}, \Pi\bar{\cup}\{(\texttt{tm}_1, \texttt{s})\})}$$

$$\frac{\begin{array}{c}\Omega \vdash_v \texttt{tm}_1 \,:\, \mathbf{ptr}(\mathbf{P}) \qquad \Omega \vdash_e \texttt{tm}_2 \,:\, \mathbf{ptr}(\mathbf{P}) \\ \Pi(\texttt{tm}_1) = \texttt{s}_1 \qquad \Pi(\texttt{tm}_2) = \texttt{s}_2 \qquad \texttt{s} = \texttt{max}(\texttt{s}_1, \texttt{s}_2)\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{tm}_1 = \texttt{tm}_2 \,:\, (\texttt{o}, \Pi\bar{\cup}\{(\texttt{tm}_1 \texttt{s})\}\bar{\cup}\{(\texttt{tm}_2, \texttt{s})\})}$$

$$\frac{\begin{array}{c}\Omega \vdash_e \texttt{tm}_1 \,:\, \mathbf{int} \qquad \Omega \vdash_e \texttt{tm}_2 \,:\, \mathbf{int} \\ \texttt{FV}(\texttt{tm}_1) \subset dom(\Pi) \qquad \texttt{FV}(\texttt{tm}_2) \subset dom(\Pi)\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{tm}_1 > \texttt{tm}_2 \,:\, (\texttt{o}, \Pi)}$$

$$\frac{\Xi; \Omega; \Pi \vdash \texttt{Pa} \,:\, (\texttt{o}, \Pi)}{\Xi; \Omega; \Pi \vdash\ \texttt{not Pa} \,:\, (\texttt{o}, \Pi)}$$

$$\frac{\begin{array}{l}\Xi(\texttt{struct}) = ((+, \texttt{yes}, \texttt{yes})\,\mathbf{ptr}(\mathbf{P})) \rightarrow (\texttt{m}_1\ \mathbf{t}_1) \cdots \rightarrow (\texttt{m}_n\ \mathbf{t}_n) \rightarrow \texttt{o} \\ \forall i \in [1, n], \\ \left\{\begin{array}{ll}\Omega \vdash_e \texttt{tm}_i \,:\, \mathbf{int}, \texttt{FV}(\texttt{tm}_i) \in \Pi & \texttt{m}_i\ \mathbf{t}_i = +\,\mathbf{int} \\ \Omega \vdash_e \texttt{tm}_i \,:\, \mathbf{ptr}(\mathbf{P}), \Pi(\texttt{tm}_i) \leq \texttt{s}_1 & \texttt{m}_i\ \mathbf{t}_i = (+, \texttt{s}_1, s_2)\,\mathbf{ptr}(\mathbf{P}) \\ \Omega \vdash_v \texttt{tm}_i \,:\, \mathbf{t}_i & \texttt{m}_i = - \text{ or } (-, \texttt{s}_1, s_2)\end{array}\right. \\ \begin{array}{lcl}\Pi' & = & \Pi \cup \{\texttt{FV}(\texttt{tm}_j) \mid m_j t_j = -\,\mathbf{int}\} \\ & & \bar{\cup}\{(\texttt{tm}_k, \texttt{s}_2) \mid m_k = (-, \texttt{s}_1, \texttt{s}_2)\} \\ & & \bar{\cup}\{(\texttt{tm}_i, \texttt{yes}) \mid m_i = (+, \texttt{no}, \texttt{yes})\}\end{array}\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{struct tm tm}_1 \cdots \texttt{tm}_n \,:\, (\texttt{o}, \Pi')}$$

$$\frac{}{\Xi; \Omega; \Pi \vdash \mathbf{P} \,:\, (\Xi(\mathbf{P}), \Pi)}$$

$$\frac{\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, ((-\,\mathbf{int}) \rightarrow \texttt{pt}, \Pi') \qquad \Omega \vdash_v \texttt{tm} \,:\, \mathbf{int}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi' \cup \texttt{FV}(\texttt{tm}))}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, ((+\,\mathbf{int}) \rightarrow \texttt{pt}, \Pi') \\ \texttt{FV}(\texttt{tm}) \subset dom(\Pi) \qquad \Omega \vdash_v \texttt{tm} \,:\, \mathbf{int}\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi')}$$

$$\frac{\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, ((*\,\mathbf{int}) \rightarrow \texttt{pt}, \Pi') \qquad \Omega \vdash_v \texttt{tm} \,:\, \mathbf{int}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi')}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, (((-, \texttt{no}, \texttt{no})\,\mathbf{ptr}(\mathbf{P})) \rightarrow \texttt{pt}, \Pi') \\ \Omega \vdash_v \texttt{tm} \,:\, \mathbf{ptr}(\mathbf{P})\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi'\bar{\cup}\{(\texttt{FV}(\texttt{tm}), \texttt{no})\})}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, (((-, \texttt{yes}, \texttt{yes})\,\mathbf{ptr}(\mathbf{P})) \rightarrow \texttt{pt}, \Pi') \\ \Omega \vdash_v \texttt{tm} \,:\, \mathbf{ptr}(\mathbf{P})\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi' \cup \{(\texttt{tm}, \texttt{yes})\})}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, (((+, \texttt{no}, \texttt{no})\,\mathbf{ptr}(\mathbf{P})) \rightarrow \texttt{pt}, \Pi') \\ \Omega \vdash_v \texttt{tm} \,:\, \mathbf{ptr}(\mathbf{P}) \qquad \texttt{FV}(\texttt{tm}) \in dom(\Pi)\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi')}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, (((+, \texttt{yes}, \texttt{yes})\,\mathbf{ptr}(\mathbf{P})) \rightarrow \texttt{pt}, \Pi') \\ \Omega \vdash_v \texttt{tm} \,:\, \mathbf{ptr}(\mathbf{P}) \qquad \Pi(\texttt{tm}) = \texttt{yes}\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi')}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, (((+, \texttt{no}, \texttt{yes})\,\mathbf{ptr}(\mathbf{P})) \rightarrow \texttt{pt}, \Pi') \\ \Omega \vdash_v \texttt{tm} \,:\, \mathbf{ptr}(\mathbf{P}) \qquad \texttt{FV}(\texttt{tm}) \in \Pi\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi'\bar{\cup}\{(\texttt{tm}, \texttt{yes})\})}$$

$$\frac{\begin{array}{c}\Xi; \Omega; \Pi \vdash \texttt{Pu} \,:\, (((*, \texttt{no}, \texttt{no})\,\mathbf{ptr}(\mathbf{P})) \rightarrow \texttt{pt}, \Pi') \\ \Omega \vdash_v \texttt{tm} \,:\, \mathbf{ptr}(\mathbf{P})\end{array}}{\Xi; \Omega; \Pi \vdash \texttt{Pu tm} \,:\, (\texttt{pt}, \Pi')}$$

$$\frac{\Xi; \Omega; \Pi \vdash \texttt{L} \,:\, (\texttt{o}, \Pi') \quad \Xi; \Omega; \Pi' \vdash \texttt{F} \,:\, (\texttt{o}, \Pi'')}{\Xi; \Omega; ; \Pi \vdash \texttt{L}, \texttt{F} \,:\, (\texttt{o}, \Pi'')}$$

- $\Pi'' < \Pi'$ iff
  - $\forall \texttt{var} \in dom(\Pi'), \texttt{var} \in dom(\Pi'')$
  - $\forall (\texttt{tm}, \texttt{yes}) \in \Pi', (\texttt{tm}, \texttt{yes}) \in \Pi'$

- $\forall(\texttt{tm},\texttt{no}) \in \Pi', (\texttt{tm},\texttt{no}) \in \Pi''$, or $(\texttt{tm},\texttt{yes}) \in \Pi''$

$\boxed{\Xi \vdash \texttt{I}\ \mathsf{OK}}$

$$\frac{\begin{array}{c}\Xi(\mathbf{P}) = \texttt{pt} \quad \Omega = \texttt{infer}(\mathbf{P}\ \overline{x}) \\ \Pi = \{x_j | \texttt{pt}_j = +\,\textbf{int}\} \cup \{(x_j, \texttt{s}_{in}) | \texttt{pt}_j = (+, \texttt{s}_{in}, \texttt{s}_{out})\ \textbf{ptr}(\mathbf{P})\} \\ \Xi;\Omega;\Pi \vdash \mathbf{P}\ \overline{x}\ :\ (\texttt{o}, \Pi') \\ \forall i \in [1,n], \quad \Omega' = \texttt{infer}(\texttt{F}_i) \\ \Xi;\Omega',\Omega;\Pi \vdash \texttt{F}_i\ :\ (\texttt{o},\Pi'') \qquad \Pi'' < \Pi'\end{array}}{\Xi \vdash ((\texttt{F}_1;\cdots;\texttt{F}_n) \multimap \mathbf{P}\ \overline{x})\ \mathsf{OK}}$$

$\boxed{\Xi \vdash \texttt{I}\ \texttt{well}-\texttt{formed}}$

$$\frac{\begin{array}{c}\exists i \in [1,n] \text{ such that } \forall j \in [1,i], \\ \text{and } \forall \texttt{L} \in \texttt{F}_j, \texttt{L} = \texttt{A} \text{ or } \texttt{L} = \texttt{Ps} \\ \forall k \in [i+1,n], \text{ and } \texttt{F}_k = \texttt{L}_1, \cdots \texttt{L}_m \\ \text{if } \exists s \in [1,m], \text{ such that } \texttt{L}_s = \texttt{Pu} \text{ and} \\ \forall t \in [1,s), \texttt{L}_s = \texttt{A} \text{ or } \texttt{Ps}, \\ \text{then } \exists y \in [1,s) \text{ such that } \texttt{L}_y = \texttt{Ps} \\ \Xi \vdash \texttt{I}\ \mathsf{OK}\end{array}}{\Xi \vdash (\texttt{F}_1;\cdots;\texttt{F}_n) \multimap \mathbf{P}\ \overline{\texttt{tm}}\ \texttt{well}-\texttt{formed}}$$

$\boxed{\vdash SS\ :\ (\Lambda;\Upsilon)}$

$$\frac{\begin{array}{c}\vdash \overline{\texttt{pdecl}}\ :\ \Xi \qquad \forall \texttt{I}_i \in \overline{\texttt{I}},\ \Xi \vdash \texttt{I}_i\ \texttt{well}-\texttt{formed} \\ \forall \texttt{Ax}_i \in \overline{\texttt{Ax}},\ \Xi \vdash \texttt{Ax}_i\ \mathsf{OK} \\ \Xi \vdash \texttt{F}\ \multimap\ \mathbf{P}\ x\ \mathsf{OK}\end{array}}{\begin{array}{l}\vdash \mathbf{P}\{\overline{\texttt{pdecl}}. \\ \quad \texttt{F}\ \multimap\ \mathbf{P}\ x. \\ \quad \overline{\texttt{I}}. \\ \quad \overline{\texttt{Ax}}\}\ : \\ \quad ((\mathbf{P}\ :\ \Xi);\ ((\texttt{F}\ \multimap\ \mathbf{P}\ x), \overline{\texttt{I}}, \overline{\texttt{Ax}}))\end{array}}$$

C. Summary of System Requirements

Closed Shape

$\mathbf{P}$ describes a cloded shape, if for all $\texttt{H}$ such that $\texttt{H} \vDash \mathbf{P}(l)$, $\texttt{H} = \texttt{H}_1 \uplus \cdots \uplus \texttt{H}_k$, and for all $i \in [1,k]$, there exists $\texttt{v}, \texttt{v}_1, \cdots, \texttt{v}_n$ such that $\texttt{H}_i \vDash\_\ \texttt{struct}\ \texttt{v}\ (\texttt{v}_1, \cdots, \texttt{v}_n)\ \forall \texttt{pt}_i = \texttt{m}\ \textbf{ptr}(\mathbf{P})$, $\texttt{v}_i = 0$ or $\texttt{v}_i \in d\bar{o}m(\texttt{H})$. where $\Lambda(\mathbf{P}) = \Xi$, $\Xi(\texttt{struct}) = \texttt{pt}$, n = $\texttt{size}(\mathbf{P},\ \texttt{struct})$

Soundness of Axioms Υ_A is sound with regard to Υ_I if for all $\texttt{Ax} \in \Upsilon_A$, $\emptyset \vDash^{\Upsilon_I} \texttt{Ax}$.

Uniqueness of Shape Matching

If $\Xi;\Omega;\Pi \vdash \texttt{L}\ :\ (\texttt{o},\Pi')$, and $\forall x \in dom(\Pi).\ x \in dom(\sigma)$ and $\texttt{H}_1 \vDash \sigma_1(\texttt{L})$, and $\texttt{H}_2 \vDash \sigma_2(\texttt{L})$, and $\sigma \subseteq \sigma_1$, and $\sigma \subseteq \sigma_2$, and $\texttt{H}_1 \subset \texttt{H}$, and $\texttt{H}_2 \subset \texttt{H}$, then $\texttt{H}_1 = \texttt{H}_2$, and $\sigma_1 = \sigma_2$.

D. Summary of the Lemmas in the Logical System

Lemma 6 (Soundness of Logical Deduction)
If $\Upsilon = \Upsilon_I, \Upsilon_A$ such that Υ_A is sound with regard to Υ_I, and $\Omega\,|\,\Upsilon;\Delta \implies \texttt{F}$, and σ is a grounding substitution for Ω, and $\texttt{H} \vDash^{\Upsilon_I} \sigma(\Delta)$, then $\texttt{H} \vDash^{\Upsilon_I} \texttt{F}$.

Theorem 7 (Termination of MP)
If for all $\texttt{I} \in \Upsilon$, $\vdash \texttt{I}\ \texttt{well}-\texttt{formed}$, then $\texttt{MP}(\Upsilon)$ always terminates.

Proof: By induction on the size of $(dom(\texttt{H}) - S)$. ■

$\boxed{\Pi\,[\,\texttt{tm}\,/\,\texttt{var}\,] = \Pi'}$

$(\Pi, \texttt{var})\,[\,\texttt{tm}\,/\,\texttt{var}\,] = \Pi \cup \texttt{FV}(\texttt{tm})$
$(\Pi, (\texttt{var},\texttt{s}))\,[\,\texttt{tm}\,/\,\texttt{var}\,] = \Pi \dot\cup \{(\texttt{tm},\texttt{s})\}$
$\Pi\,[\,\texttt{tm}\,/\,\texttt{var}\,] = \Pi$ if $\texttt{var} \notin dom(\Pi)$

Lemma 8 (Substitution)
If $\Xi;\Omega,\texttt{var}{:}\mathbf{t};\Pi \vdash \texttt{F}\ :\ (\texttt{pt},\Pi')$, and $\Omega' \vdash_v \texttt{tm}\ :\ \mathbf{t}$ then $\Xi;\Omega \cup \Omega';\Pi\,[\,\texttt{tm}\,/\,\texttt{var}\,]\ \vdash\ \texttt{F}\,[\,\texttt{tm}\,/\,\texttt{var}\,]\ :\ (\texttt{pt},\Pi_1)$ and $\Pi_1 < \Pi'\,[\,\texttt{tm}\,/\,\texttt{var}\,]$

Theorem 9 (Correctness of MP)
If $\Xi;\Omega;\Pi \vdash \texttt{F}\ :\ (\texttt{o},\Pi')$, and $\forall x \in dom(\Pi).\ x \in dom(\sigma)$, and $S \subset dom(\texttt{H})$ then

- *either $\texttt{MP}(\texttt{H};S;\texttt{F};\sigma) = (S',\sigma')$ and $S' \subset dom(\texttt{H})$, $\sigma \subset \sigma'$, and $\texttt{H}' \vDash \sigma'(\texttt{F})$, and $dom(\texttt{H}') = (S' - S)$, $\forall x \in dom(\Pi').\ x \in dom(\sigma')$,*
- *Or $\texttt{MP}(\texttt{H};S;\texttt{F};\sigma) = \texttt{NO}$, and there exists no $\texttt{H}'$, $dom(\texttt{H}') \subset (dom(\texttt{H}) - S)$, and σ', $\sigma \subset \sigma'$, such that $\texttt{H}' \vDash \sigma'(\texttt{F})$*

Lemma 10 (Safety of MP)
If for all $\texttt{I} \in \Upsilon$, $\vdash \texttt{I}\ \texttt{well}-\texttt{formed}$, $\mathbf{P}$ is a closed shape, and $\Xi = \Lambda(\mathbf{P})$, $\texttt{H}_1 \vDash \mathbf{P}(l)$, $\Xi;\Omega;\Pi \vdash \texttt{F}\ :\ (\texttt{o},\Pi')$, and $\forall x \in dom(\Pi).\ x \in dom(\sigma)$, and $S \subset dom(\texttt{H}_1)$, and $\forall \texttt{tm}$ such that $\Pi(\texttt{tm}) = \texttt{yes}$, $\sigma(\texttt{tm}) \in d\bar{o}m(\texttt{H}_1)$ or $\sigma'(\texttt{tm}) = 0$ then

- *either $\texttt{MP}(\Upsilon)(\texttt{H}_1 \uplus \texttt{H}_2;S;\texttt{F};\sigma) = (S',\sigma')$, and $\texttt{MP}$ will not access location l if $l \notin dom(\texttt{H}_1)$, and $\forall x \in dom(\Pi').\ x \in d\bar{o}m(\sigma')$, and $\forall \texttt{tm}$ such that $\Pi'(\texttt{tm}) = \texttt{yes}$, then $\sigma'(\texttt{tm}) \in d\bar{o}m(\texttt{H}_1)$ or $\sigma'(\texttt{tm}) = 0$*
- *Or $\texttt{MP}(\Upsilon)(\texttt{H}_1 \uplus \texttt{H}_2;S;\texttt{F};\sigma) = \texttt{NO}$, and $\texttt{MP}$ will not access location l if $l \notin dom(\texttt{H}_1)$.*

E. Summary of Operational Semantics

Runtime Constructs

Runtime stmt	$\texttt{stmt}$	$::=$	$\texttt{fail} \mid \texttt{halt}$
Eval Ctxt	$\mathcal{C}_e$	$::=$	$[\,]_e \mid \mathcal{C}_e + e \mid v + \mathcal{C}_e \mid -\mathcal{C}_e$
	$\mathcal{C}_{stmt}$	$::=$	$[\,]_{stmt} \mid \mathcal{C}_{stmt}\ ;\ \texttt{stmt} \mid \$x\ :=\ \mathcal{C}_e$
			$\mid \$s\ :=\ f\ (\ v_1, \cdots v_k, \mathcal{C}_e, e_{k+2}, \cdots\ )$
	$\mathcal{C}_{call}$	$::=$	$[\,]_{stmt} \mid \mathcal{C}_{call}\ ;\ \texttt{stmt}$
Environment	E	$::=$	$\cdot \mid E, \$x \mapsto n \mid E, \$s \mapsto n$
Env Stack	Es	$::=$	$\bullet \mid E\ \triangleright\ Es$
Control Stack	S	$::=$	$\bullet \mid \mathcal{C}_{stmt}[\$s\ :=\ \bullet]\ \triangleright\ S$

$\boxed{[\![\, cc \,]\!]_E = (F,\sigma)}$

$[\![\, \texttt{A} \,]\!]_E = (E(\texttt{A}), \cdot)$
$[\![\, \$s?[\texttt{root}\ x, \texttt{F}] \,]\!]_E = (E(\texttt{F}), \{E(\$s)/x\})$
$[\![\, \$s{:}[\texttt{root}\ x, \texttt{F}] \,]\!]_E = (E(\texttt{F}), \{E(\$s)/x\})$
$[\![\, cc_1, cc_2 \,]\!]_E = ((\texttt{F}_1,\texttt{F}_2), \sigma_1 \cup \sigma_2)$ if $[\![\, cc_i \,]\!]_E = (\texttt{F}_i, \sigma_i)$

Operational Semantics for Expressions

var	$(E;\$x) \longmapsto E(\$x)$
sum	$(E; v_1 + v_2) \longmapsto v$ where $v = v_1 + v_2$
ctx	$(E; \mathcal{C}_e[e]) \longmapsto \mathcal{C}_e[e']$ if $(E;e) \longmapsto e'$

Operational Semantics for Statements

$(E;\mathtt{H};\mathtt{stmt}) \longmapsto (E';\mathtt{H}';\mathtt{stmt}')$	
seq	$(E;\mathtt{H};(\mathtt{skip}\ ;\ \mathtt{stmt})) \longmapsto (E;\mathtt{H};\mathtt{stmt})$
assign-exp	$(E;\mathtt{H};\$x\ :=\ v) \longmapsto (E[\$x := v];\mathtt{H};\mathtt{skip})$
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	$(E;\mathtt{H};\$s\ :=\ \{\overline{x{:}\mathbf{ptr}(\mathbf{P})}\}\ [\mathtt{root}\ (v), \mathtt{F}])$ $\longmapsto (E[\$s := v'];\mathtt{H}';\mathtt{skip})$ where $(v', \mathtt{H}') =$ $\mathtt{CreateShape}(\mathtt{H}, \{\overline{x{:}\mathbf{ptr}(\mathbf{P})}\}\ [\mathtt{root}\ (v), \mathtt{F}], \mathbf{P})$
If-t	$(E;\mathtt{H};\mathtt{if}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2)$ $\longmapsto (E;\mathtt{H};\sigma(\mathtt{stmt}_1))$ if $[\![\ \mathtt{cc}\ ]\!]_E = (\mathtt{F}, \sigma')$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma') = (SL;\sigma)$
If-f	$(E;\mathtt{H};\mathtt{if}\ \{\overline{x{:}\mathbf{t}}\}\ cc\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2)$ $\longmapsto (E;\mathtt{H};\mathtt{stmt}_2)$ if $[\![\ cc\ ]\!]_E = (\mathtt{F}, \sigma)$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma) = \mathtt{NO}$
while-t	$(E;\mathtt{H};\mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt})$ $\longmapsto (E;\mathtt{H};(\sigma(\mathtt{stmt}_1)\ ;\ \mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt}))$ if $[\![\ \mathtt{cc}\ ]\!]_E = (\mathtt{F}, \sigma')$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma') = (SL;\sigma)$
while-f	$(E;\mathtt{H};\mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt}) \longmapsto (E;\mathtt{H};\mathtt{skip})$ if $[\![\ \mathtt{cc}\ ]\!]_E = (\mathtt{F}, \sigma)$ and $\mathtt{MP}(\mathtt{H};\mathtt{F};\emptyset;\sigma) = \mathtt{NO}$
switch-t	$(E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \{\overline{x{:}\mathbf{t}}\}\ ([\mathtt{root}\ (x_i), \mathtt{F}] \to \mathtt{stmt}_k)$ $\mid \mathtt{bs}\ )$ $\longmapsto (E;\mathtt{H};\sigma(\mathtt{stmt}_k))$ if $\mathtt{MP}(\mathtt{H};E(\mathtt{F});\emptyset;\{E(\$s)/x_i\}) = (SL;\sigma)$
switch-f	$(E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \{\overline{x{:}\mathbf{t}}\}\ ([\mathtt{root}\ (x_i), \mathtt{F}] \to \mathtt{stmt}_k)$ $\mid\ \mathtt{bs}\ )$ $\longmapsto (E;\mathtt{H};\mathtt{switch}\ x\ \mathtt{of}\ \mathtt{bs}\ )$ if $\mathtt{MP}(\mathtt{H};E(\mathtt{F});\emptyset;\{E(\$s)/x_i\}) = \mathtt{NO}$
fail	$(E;\mathtt{H};\mathtt{switch}\ \$s\ \mathtt{of}\ \{\overline{x{:}\mathbf{t}}\}\ ([\mathtt{root}\ (x_i), \mathtt{F}] \to \mathtt{stmt}_k))$ $\longmapsto (E;\mathtt{H};\mathtt{fail})$ if $\mathtt{MP}(\mathtt{H};E(\mathtt{F});\emptyset;\{E(\$s)/x_i\}) = \mathtt{NO}$
ctxt-e	$(E;\mathtt{H};\mathcal{C}_{stmt}[e]) \longmapsto (E';\mathtt{H}';\mathcal{C}_{stmt}[e'])$ if $(E;e) \longmapsto e'$
ctxt-fail	$(E;\mathtt{H};\mathcal{C}_{stmt}[\mathtt{stmt}]) \longmapsto (E';\mathtt{H}';\mathtt{fail})$ if $(E;\mathtt{H};\mathtt{stmt}) \longmapsto (E';\mathtt{H}';\mathtt{fail})$
ctxt-stmt	$(E;\mathtt{H};\mathcal{C}_{stmt}[\mathtt{stmt}]) \longmapsto (E';\mathtt{H}';\mathcal{C}_{stmt}[\mathtt{stmt}'])$ if $(E;\mathtt{H};\mathtt{stmt}) \longmapsto (E';\mathtt{H}';\mathtt{stmt}')$

Operational Semantics for Function Bodies

$(E;\mathtt{H};S;\mathtt{fb}) \longmapsto (E';\mathtt{H}';S';\mathtt{fb}')$	
fun-call	$(E;\mathtt{H};S;(\mathcal{C}_{stmt};\mathtt{fb})[\$s\ :=\ f\ (\ v_1 \ldots v_n\ )])$ $\longmapsto (E_f;\mathtt{H};(\mathcal{C}_{stmt};\mathtt{fb})[\$s\ :=\ \bullet]\ \triangleright\ S;\mathtt{fb}_f)$ if $\Phi(f) = ([x_1 \ldots x_n]ldecls;\mathtt{fb}_f)$ $E_f = (ldecls, x_1 \mapsto E(v_1), \cdots, x_n \mapsto E(v_n))\ \triangleright\ E.$
fun-ret	$(E\ \triangleright\ Es;\mathtt{H};(\mathcal{C}_{stmt};\mathtt{fb})[\$s\ :=\ \bullet]\ \triangleright\ S;\mathtt{return}\ \$s_1)$ $\longmapsto (Es[\$s := E(\$s_1)];\mathtt{H};S;(\mathcal{C}_{stmt};\mathtt{fb})[\mathtt{skip}])$
halt	$(\bullet;\mathtt{H};\bullet;\mathtt{return}\ v) \longmapsto (\bullet;\mathtt{H};\bullet;\mathtt{halt})$
fail	$(E;\mathtt{H};S;(\mathtt{fail}\ ;\ \mathtt{fb})) \longmapsto (\bullet;\mathtt{H};\bullet;\mathtt{halt})$
skip	$(E;\mathtt{H};S;(\mathtt{skip}\ ;\ \mathtt{fb})) \longmapsto (E;\mathtt{H};S;\mathtt{fb})$
context-stmt	$(E;\mathtt{H};S;(\mathtt{stmt}\ ;\ \mathtt{fb})) \longmapsto (E;\mathtt{H};S;(\mathtt{stmt}'\ ;\ \mathtt{fb}))$ if $(E;\mathtt{H};\mathtt{stmt}) \longmapsto (E';\mathtt{H}';\mathtt{stmt}')$

F. Summary of Typing Rules

$$\boxed{\Omega \vdash_e e\ :\ \mathbf{t}}$$

$$\frac{}{\Omega \vdash_e \mathtt{var}\ :\ \Omega(\mathtt{var})} \qquad \frac{}{\Omega \vdash_e n\ :\ \mathbf{int}} \qquad \frac{}{\Omega \vdash_e n\ :\ \mathbf{ptr}(\mathbf{P})}$$

$$\boxed{\Omega \vdash_v e\ :\ \mathbf{t}}$$

$$\frac{\Omega \vdash_e e\ :\ \mathbf{t} \quad e = x \text{ or } e = \$x \text{ or } e = n}{\Omega \vdash_v e\ :\ \mathbf{t}}$$

$$\boxed{\Omega;\Gamma \vdash_{\mathtt{cc}} \mathtt{cc}\ :\ (\Gamma';\Theta;\Delta)}$$

$$\frac{}{\Omega;\Gamma \vdash_{\mathtt{cc}} \mathtt{A}\ :\ (\Gamma;\ \cdot\ ;\ \cdot)}$$

$$\frac{\Gamma(\$s) = \mathbf{P} \qquad \Omega \mid \Upsilon;\mathtt{F} \Longrightarrow \mathbf{P}(y)}{\Omega;\Gamma \vdash \$s?[\mathtt{root}\ y, \mathtt{F}]\ :\ (\Gamma;\emptyset;\emptyset)}$$

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

$$\frac{\Omega;\Gamma_1 \vdash \mathtt{cc}_1\ :\ (\Gamma_1';\Theta_1';\Delta_1') \quad \Omega;\Gamma_2 \vdash \mathtt{cc}_2\ :\ (\Gamma_2';\Theta_2';\Delta_2')}{\Omega;\Gamma_1, \Gamma_2 \vdash \mathtt{cc}_1, \mathtt{cc}_2\ :\ (\Gamma_1', \Gamma_2';\Theta_1', \Theta_2';\Delta_1', \Delta_2')}$$

$$\boxed{\Omega;\Gamma;\Pi \vdash \mathtt{cc}\ :\ \Pi'}$$

$$\frac{\cdot;\Omega;\Pi \vdash \mathtt{A}\ :\ \Pi'}{\Omega;\Gamma;\Pi \vdash \mathtt{A}\ :\ \Pi'}$$

$$\frac{\Gamma(\$s) = \mathbf{P} \quad \Lambda(P) = \Xi \quad \Xi;\Omega;\Pi\bar{\cup}\{x{:}\mathtt{yes}\} \vdash \mathtt{F}\ :\ (\mathtt{o}, \Pi')}{\Omega;\Gamma;\Pi \vdash \$s?[\mathtt{root}\ x, \mathtt{F}]\ :\ \Pi'}$$

$$\frac{\Gamma(\$s) = \mathbf{P} \quad \Lambda(P) = \Xi \quad \Xi;\Omega;\Pi\bar{\cup}\{x{:}\mathtt{yes}\} \vdash \mathtt{F}\ :\ (\mathtt{o}, \Pi')}{\Omega;\Gamma;\Pi \vdash \$s\ :\ [\mathtt{root}\ x, \mathtt{F}]\ :\ \Pi'}$$

$$\frac{\Omega;\Gamma;\Pi \vdash \mathtt{cc}_1\ :\ \Pi' \quad \Omega;\Gamma;\Pi' \vdash \mathtt{cc}_2\ :\ \Pi''}{\Omega;\Gamma;\Pi \vdash \mathtt{cc}_1, \mathtt{cc}_2\ :\ \Pi''}$$

$$\boxed{\Omega;\Gamma;\Theta \vdash_a \mathtt{arg}\ :\ (\mathbf{t};\Gamma';\Theta')}$$

$$\frac{\Omega;\Gamma;\Theta \vdash_e e\ :\ \mathbf{t}}{\Omega;\Gamma;\Theta \vdash_a e\ :\ (\mathbf{t};\Gamma;\Theta)} \qquad \frac{\Gamma = \Gamma', \$s{:}\mathbf{P}}{\Omega;\Gamma;\Theta \vdash_a \$s\ :\ (\mathbf{P};\Gamma';\Theta, \$s{:}\mathbf{P})}$$

$$\boxed{\Omega \vdash \mathtt{ldecl}\ :\ (\Omega';\Theta)}$$

$$\frac{\Omega \vdash_e e\ :\ \mathbf{int}}{\Omega \vdash \mathbf{int}\ \ \$x\ :=\ e\ :\ (\Omega, \$x{:}\mathbf{int};\cdot)} \qquad \frac{}{\Omega \vdash \mathbf{P}\ \ \$s\ :\ (\Omega;\$s{:}\mathbf{P})}$$

$$\frac{\Omega \vdash \mathtt{ldecl}\ :\ (\Omega_1;\Theta_1) \quad \Omega_1 \vdash \overline{\mathtt{ldecl}}\ :\ (\Omega_2;\Theta_2) \quad dom(\Theta_1) \cap dom(\Theta_2) = \emptyset}{\Omega \vdash \mathtt{ldecl}\ \ \overline{\mathtt{ldecl}}\ :\ (\Omega_2;(\Theta_1, \Theta_2))}$$

$$\boxed{\Omega;\Gamma;\Theta;\Delta \vdash \mathtt{stmt}\ :\ (\Gamma';\Theta';\Delta')}$$

$$\frac{}{\Omega;\Gamma;\Theta;\Delta \vdash \mathtt{skip}\ :\ (\Gamma;\Theta;\Delta)}\ \textit{skip}$$

$$\frac{\begin{array}{c}\Omega;\Gamma;\Theta;\Delta \vdash \mathtt{stmt}_1\ :\ (\Gamma';\Theta';\Delta') \\ \Omega;\Gamma';\Theta';\Delta' \vdash \mathtt{stmt}_2\ :\ (\Gamma'';\Theta'';\Delta'')\end{array}}{\Omega;\Gamma;\Theta;\Delta \vdash \mathtt{stmt}_1\ ;\ \mathtt{stmt}_2\ :\ (\Gamma'';\Theta'';\Delta'')}\ \textit{seq}$$

$$\frac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad dom(\Omega') \subset \mathtt{FV}(\mathtt{cc}) \quad \Omega', \Omega;\Gamma \vdash \mathtt{cc}\ :\ (\Gamma', \Theta', \Delta') \\ \Omega', \Omega;\Gamma';\Theta, \Theta';\Delta, \Delta' \vdash \mathtt{stmt}_1\ :\ (\Gamma'';\Theta'';\Delta'') \\ \Omega;\Gamma;\Theta;\Delta \vdash \mathtt{stmt}_2\ :\ (\Gamma'';\Theta'';\Delta'') \\ \Pi = \mathtt{ground}(\Omega) \qquad \Omega', \Omega;\Gamma;\Pi \vdash \mathtt{cc}\ :\ \Pi' \\ \forall x_i \in dom(\Omega'), x_i \in dom(\Pi')\end{array}}{\begin{array}{r}\Omega;\Gamma;\Theta;\Delta \vdash \mathtt{if}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2 \\ :\ (\Gamma'';\Theta'';\Delta'')\end{array}}\ \textit{if}$$

$$\frac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad dom(\Omega') \subset \mathtt{FSV}(\mathtt{cc}) \\ \Omega', \Omega;\Gamma \vdash \mathtt{cc}\ :\ (\Gamma', \Theta', \Delta') \\ \Omega', \Omega;\Gamma';\Theta, \Theta';\Delta, \Delta' \vdash \mathtt{stmt}\ :\ (\Gamma;\Theta;\Delta) \\ \Pi = \mathtt{ground}(\Omega) \qquad \Omega', \Omega;\Gamma;\Pi \vdash \mathtt{cc}\ :\ \Pi' \\ \forall x_i \in dom(\Omega'), x_i \in dom(\Pi')\end{array}}{\Omega;\Gamma;\Theta;\Delta \vdash \mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathtt{cc}\ \mathtt{do}\ \mathtt{stmt}\ :\ (\Gamma;\Theta;\Delta)}\ \textit{while}$$

$$\frac{\Omega \vdash_e \$x\ :\ \mathbf{t} \quad \Omega \vdash_e e\ :\ \mathbf{t}}{\Omega;\Gamma;\Theta;\Delta \vdash \$x\ :=\ e\ :\ (\Gamma;\Theta;\Delta)}\ \textit{assign-exp}$$

$$\frac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{ptr}(\mathbf{P})} \qquad \Theta = \Theta' \cup (\$s : \mathbf{P}) \\ \Omega', \Omega \,|\, \Upsilon; \mathtt{F} \Longrightarrow \mathbf{P}(v) \\ \Delta_x = \{\mathtt{struct}\ x_i\ \overline{e} \,|\, \mathtt{struct}\ x_i\ \overline{e} \in \mathtt{F}\} \\ \Delta = \Delta', \Delta'' \qquad \mathtt{F} = \Delta_x \uplus \Delta_F \\ \forall \mathtt{Pu}, \mathtt{Pu} \in \Delta'' \text{ iff } \mathtt{Pu} \in \Delta_F \\ \forall \mathtt{Ps} = \mathtt{struct\ tm}\ \overline{e}, \mathtt{Ps} \in \Delta'' \text{ iff } \mathtt{struct\ tm}\ \overline{e'} \in \Delta_F \end{array}}{\begin{array}{l}\Omega; \Gamma; \Theta; \Delta \vdash \$s \ := \ \{\overline{x{:}\mathbf{ptr}(\mathbf{P})}\}[\mathtt{root}\ (v), \mathtt{F}] \\ \qquad\qquad\qquad : \ (\Gamma' \cup (\$s{:}\mathbf{P}); \Theta'; \Delta')\end{array}} \ \textit{assign-shape}$$

$$\frac{\text{For all i, } 1 \le i \le n, \quad \Omega; \Gamma; \Theta; \Delta \vdash_{\$s} \mathtt{b}_{\ i} \ : \ (\Gamma'; \Theta'; \Delta')}{\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{switch}\ \$s\ \mathtt{of\ bs} \ : \ (\Gamma'; \Theta'; \Delta')} \ \textit{switch}$$

$$\frac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad \Gamma(\$s) = \mathbf{P} \qquad \Xi = \Lambda(\mathbf{P}) \\ \Xi; \Omega', \Omega; \Gamma; \mathtt{ground}(\Omega)\bar{\cup}\{x_i{:}\mathtt{yes}\} \vdash \mathtt{F} \ : \ \Pi \\ \forall x_i \in \mathit{dom}(\Omega'), x_i \in \mathit{dom}(\Pi) \\ \Omega', \Omega \,|\, \Upsilon; \mathtt{F} \Longrightarrow \mathbf{P}(x_i) \\ \Omega', \Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma'; \Theta'; \Delta')\end{array}}{\Omega; \Gamma; \Theta; \Delta \vdash_{\$s} \{\overline{x{:}\mathbf{t}}\}\ ?[\mathtt{root}\ (x_i), \mathtt{F}] \to \mathtt{stmt} \ : \ (\Gamma'; \Theta'; \Delta')} \ \textit{pat-?}$$

$$\frac{\begin{array}{c}\Omega' = \overline{x{:}\mathbf{t}} \qquad \Gamma = \Gamma', \$s{:}\mathbf{P} \\ \Xi = \Lambda(\mathbf{P}) \qquad \mathtt{Shp} = \mathtt{root}\ (x_i), \mathtt{F}_A, \mathtt{F} \\ \Xi; \Omega; \Omega'; \Gamma; \mathtt{ground}(\Omega)\bar{\cup}\{x_i{:}\mathtt{yes}\} \vdash \mathtt{F} \ : \ \Pi \\ \forall x_i \in \mathit{dom}(\Omega'), x_i \in \mathit{dom}(\Pi) \\ \Omega', \Omega \,|\, \Upsilon; \mathtt{F}_A, \mathtt{F} \Longrightarrow \mathbf{P}(x_i) \qquad \mathtt{FV}(\mathtt{F}) \cap \Omega_{\$} = \emptyset \\ \Omega', \Omega; \Gamma'; \Theta, \$s{:}\mathbf{P}; \Delta, \mathtt{F} \vdash \mathtt{stmt} \ : \ (\Gamma''; \Theta'; \Delta')\end{array}}{\Omega; \Gamma; \Theta; \Delta \vdash_{\$s} \{\overline{x{:}\mathbf{t}}\}\ {:}[\mathtt{Shp}] \to \mathtt{stmt} \ : \ (\Gamma''; \Theta'; \Delta')} \ \textit{pat-:}$$

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

$$\frac{\begin{array}{c}\Phi(f) = (\tau_1 \times \cdots \times \tau_n \to \mathbf{P}) \\ \Omega; \Gamma; \Theta \vdash_a a_1 \ : \ (\tau_1; \Gamma_1; \Theta_1) \\ \Omega; \Gamma_1; \Theta_1 \vdash_a a_2 \ : \ (\tau_2; \Gamma_2; \Theta_2) \\ \cdots \\ \Omega; \Gamma_{n-1}; \Theta_{n-1} \vdash_a a_n \ : \ (\tau_n; \Gamma_n; \Theta_n) \\ \Theta_n = \Theta, \$s{:}\mathbf{P}\end{array}}{\Omega; \Gamma; \Theta; \Delta \vdash \$s \ := \ f\ (\ a_1, \cdots, a_n\ ) \ : \ ((\Gamma_n, \$s{:}\mathbf{P}; \Theta; \Delta)} \ \textit{fun-call}$$

$$\boxed{\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{fb} \ : \ \mathbf{P}}$$

$$\frac{\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma'; \Theta'; \Delta') \quad \Omega; \Gamma'; \Theta'; \Delta' \vdash \mathtt{fb} \ : \ \mathbf{P}}{\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt}\ ;\ \mathtt{fb} \ : \ \mathbf{P}} \ \textit{seq}$$

$$\frac{}{\Omega; \$s{:}\mathbf{P}; \Theta; \cdot \vdash \mathtt{return}\ \$s \ : \ \mathbf{P}} \ \textit{return}$$

$$\boxed{\Phi \vdash \mathtt{fdecl} \ : \ (\tau_1 \times \cdots \times \tau_n) \to \mathbf{P}}$$

$$\frac{\$x_1{:}\tau_1, \cdots \$x_n{:}\tau_n \vdash \overline{\mathtt{ldecl}} \ : \ (\Omega; \Theta) \quad \Omega; \cdot; \Theta; \cdot \vdash \mathtt{fb} \ : \ \tau_s}{\Phi \vdash \mathbf{P}\ \ f(x_1 : \tau_1, \cdots x_n : \tau_n)\,\{\,\overline{\mathtt{ldecl}}; \mathtt{fb}\,\} \ : \ ((\tau_1 \times \cdots \tau_n) \to \mathbf{P})}$$

$$\boxed{\vdash \overline{\mathtt{fdecl}} \ : \ \Phi}$$

$$\frac{\text{for all } \mathtt{fdecl}_i \in \overline{\mathtt{fdecl}}, \Phi \vdash \mathtt{fdecl} \ : \ \Phi(f)}{\vdash \overline{\mathtt{fdecl}} \ : \ \Phi}$$

$$\boxed{[\![\,\Gamma\,]\!]_E = F} \quad [\![\,\cdot\,]\!]_E = \mathtt{emp}$$

$$[\![\,\Gamma, \$s{:}\mathbf{P}\,]\!]_E = [\![\,\Gamma\,]\!]_E, \mathbf{P}(E(\$s))$$

$$\boxed{\vdash \mathsf{prog\ OK}}$$

$$\frac{\vdash \overline{\mathtt{SS}} \ : \ (\Lambda; \Upsilon) \quad \Lambda; \Upsilon \vdash \overline{\mathtt{fdecl}} \ : \ \Phi}{\vdash \overline{\mathtt{SS}} \quad \overline{\mathtt{fdecl}}\ \mathsf{OK}}$$

$$\boxed{\vdash (Es; \mathtt{H}; S; \mathtt{fb})\ \mathsf{OK}}$$

$$\frac{}{\vdash (\bullet; \mathtt{H}; \bullet; \mathtt{halt})\ \mathsf{OK}}$$

$$\frac{\vdash E \ : \ \Omega \quad \mathtt{H} \vDash [\![\,\Gamma\,]\!]_E \otimes \Delta \quad \Omega; \Gamma; \Theta; \Delta \vdash \mathtt{fb} \ : \ \tau}{\vdash (E \ \rhd \ \bullet; \mathtt{H}; \bullet; \mathtt{fb})\ \mathsf{OK}}$$

$$\frac{\begin{array}{c}\vdash E \ : \ \Omega \qquad \mathtt{H} = \mathtt{H}_1 \uplus \mathtt{H}_2 \\ \mathtt{H}_1 \vDash [\![\,\Gamma\,]\!]_E \otimes \Delta \qquad \cdot; \Gamma; \Theta; \Delta \vdash \mathtt{fb} \ : \ \mathbf{P} \\ \forall \mathtt{H}' \text{ such that } \mathtt{H}' \vDash \mathbf{P}(l) \text{ and } \mathtt{H}' \# \mathtt{H}_2 \\ \vdash (Es[\$s \ := \ l]; \mathtt{H}' \uplus \mathtt{H}_2; S; \mathcal{C}_{call}[\mathtt{skip}])\ \mathsf{OK}\end{array}}{\vdash (E \ \rhd \ Es; \mathtt{H}; \mathcal{C}_{call}[\$s \ := \ \bullet] \ \rhd \ S; \mathtt{fb})\ \mathsf{OK}}$$

G. Summary of the Lemmas in Proving Progress and Preservation Theorem

Lemma 11 (Substitution)

- *If* $\Omega, x{:}\mathbf{t} \,|\, \Upsilon; \Delta \Longrightarrow \mathtt{F}$*, and* $\vdash v \ : \ \mathbf{t}$ *then* $\Omega \,|\, \Upsilon; \Delta\,[\,v\,/\,x\,] \Longrightarrow \mathtt{F}\,[\,v\,/\,x\,]$
- *If* $\Omega, x{:}\mathbf{t} \vdash e \ : \ \mathbf{t}$*, and* $\vdash v \ : \ \mathbf{t}$*, then* $\Omega \vdash e\,[\,v\,/\,x\,] \ : \ \mathbf{t}$
- *If* $\Omega, x{:}\mathbf{t}; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma'; \Theta'; \Delta')$*, and* $\vdash v \ : \ \mathbf{t}$*, then* $\Omega; \Gamma; \Theta; \Delta\,[\,v\,/\,x\,] \vdash \mathtt{stmt}\,[\,v\,/\,x\,] \ : \ (\Gamma'; \Theta; \Delta'\,[\,v\,/\,x\,])$*,*

Lemma 12 (Unique Decomposition)

1. *If* $\Omega_{\$} \vdash e \ : \ t$ *then either* e *is a value or* $e = \mathcal{C}_e[\mathit{redex}]$ *where* $\mathit{redex} = \$x \mid v + v$
2. *If* $\Omega_{\$}; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma'; \Theta'; \Delta')$ *then either* $\mathtt{stmt} = \mathtt{skip}$ *or* $\mathtt{fail}$ *or* $\mathtt{stmt} = \mathcal{C}_{stmt}[\mathit{redex}]$
   where $\mathit{redex} = \mathtt{skip}\ ;\ \mathtt{stmt} \mid \$x \ := \ v$
   $\mid \mathtt{free}\ v \mid \$s \ := \ \{\overline{x{:}\mathbf{t}}\}\,\mathtt{Shp}$
   $\mid \mathtt{if}\ \{\overline{x{:}\mathbf{t}}\}\ \mathit{cc}\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2$
   $\mid \mathtt{while}\ \{\overline{x{:}\mathbf{t}}\}\ \mathit{cc}\ \mathtt{do\ stmt}$
   $\mid \mathtt{switch}\ x\ \mathtt{of\ bs} \mid \$x \mid v + v$
   or $\mathtt{stmt} = \mathcal{C}_{call}[x \ := \ f\ (\ v_1, \cdots, v_n\ )]$

Lemma 13 (Context properties)

1. *If* $\Omega \vdash e \ : \ t_1$ *and* $\Omega \vdash \mathcal{C}_e[e] \ : \ t_2$ *then for all* e' *such that* $\Omega \vdash e' \ : \ t_1$*,* $\Omega; \Gamma \vdash \mathcal{C}_e[e'] \ : \ t_2$
2. *If* $\Omega \vdash e \ : \ t_1$ *and* $\Omega; \Gamma; \Theta; \Delta \vdash \mathcal{C}_{stmt}[e] \ : \ (\Gamma'; \Theta'; \Delta')$ *then for all* e' *such that* $\Omega \vdash e' \ : \ t_1$*,* $\Omega; \Gamma; \Theta; \Delta \vdash \mathcal{C}_{stmt}[e'] \ : \ (\Gamma'; \Theta'; \Delta')$
3. *If* $\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma_1; \Theta_1; \Delta_1)$ *and* $\Omega; \Gamma; \Theta; \Delta \vdash \mathcal{C}_{stmt}[\mathtt{stmt}] \ : \ (\Gamma_2; \Theta_2; \Delta_2)$ *then for all* $\mathtt{stmt}'$ *such that* $\Omega; \Gamma'; \Theta'; \Delta' \vdash \mathtt{stmt}' \ : \ (\Gamma_1; \Theta_1; \Delta_1)$*,* $\Omega; \Gamma'; \Theta'; \Delta' \vdash \mathcal{C}_{stmt}[\mathtt{stmt}'] \ : \ (\Gamma_2; \Theta_2; \Delta_2)$
4. *if* $\Omega \vdash \mathcal{C}_e[e] \ : \ t$ *then there exists* t_1 *such that* $\Omega \vdash e \ : \ t_1$*.*
5. *if* $\Omega; \Gamma; \Theta; \Delta \vdash \mathcal{C}_{stmt}[e] \ : \ (\Gamma'; \Theta'; \Delta')$ *then there exists* t_1 *such that* $\Omega; \Gamma \vdash e \ : \ t_1$*.*
6. *if* $\Omega; \Gamma; \Theta; \Delta \vdash \mathcal{C}_{stmt}[\mathtt{stmt}] \ : \ (\Gamma_2, \Theta_2; \Delta_2)$ *then exists* Γ_1*,* Θ_1*,* Δ_1 *such that* $\Omega; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma_1; \Theta_1; \Delta_1)$*.*

Lemma 14 (Frame Properity)

If $(E; \mathtt{H}; \mathtt{stmt}) \ \longmapsto \ (E'; \mathtt{H}'; \mathtt{stmt}')$*, and* $\mathtt{H} \ = \ \mathtt{H}_1 \uplus \mathtt{H}_2$*, and* $\Omega_{\$}; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} \ : \ (\Gamma'; \Theta'; \Delta')$*,* $\vdash E_{\$} \ : \ \Gamma_{\$}$*,* $\mathtt{H}_1 \vDash ([\![\,\Gamma\,]\!]_E) \otimes \Delta$*, then* $(E; \mathtt{H}_1; \mathtt{stmt}) \longmapsto (E'; \mathtt{H}_1'; \mathtt{stmt}')$*, such that* $\mathtt{H}' = \mathtt{H}_1' \uplus \mathtt{H}_2$

$$\boxed{\mathsf{shape}(\mathsf{cc}, E, \Gamma) = \mathtt{F}}$$

$\mathsf{shape}(A, E, \Gamma) = \mathtt{emp}$
$\mathsf{shape}(\$s?[\mathsf{Shp}], E, \Gamma) = \mathbf{P}(l)$ if $\Gamma(\$s) = \mathbf{P}$, and $E(\$s) = l$
$\mathsf{shape}(\$s{:}[\mathsf{Shp}], E, \Gamma) = \mathbf{P}(l)$ if $\Gamma(\$s) = \mathbf{P}$, and $E(\$s) = l$
$\mathsf{shape}((cc_1, cc_2), E, \Gamma) = (F_1, F_2)$ if $\mathsf{shape}(cc_i, E, \Gamma) = F_i$

$$\boxed{\mathsf{ctx}(\mathsf{cc}, \Gamma) = \Gamma'}$$

$\mathtt{ctx}(A, E, \Gamma) = \cdot \qquad \mathtt{ctx}(\$s?[\mathtt{Shp}], \Gamma) = \$s{:}\Gamma(\$s)$
$\mathtt{ctx}(\$s{:}[\mathtt{Shp}], \Gamma) = \$s{:}\Gamma(\$s)$
$\mathtt{ctx}((cc_1, cc_2), \Gamma) = (\Gamma_1, \Gamma_2)$ if $\mathtt{ctx}(cc_i, \Gamma) = \Gamma_i$

Lemma 15 (Conjunctive Clauses)

- *If* $\Omega; \Gamma \vdash \mathtt{cc} : (\Gamma'; \Theta; \Delta)$ *and* $\mathtt{ctx}(\mathtt{cc}, \Gamma) = \Gamma_{\mathtt{cc}}$*, then* $\Gamma' = \Gamma'_1, \Gamma'_2$ *such that* $\Gamma_{cc}, \Gamma'_1 = \Gamma$*.*
- *If* $\Omega; \Gamma \vdash \mathtt{cc} : (\Gamma'; \Theta; \Delta)$*, and* $[\![\, \mathtt{cc} \,]\!]_E = (\mathtt{F}_{cc}, \sigma')$ *and* $\mathtt{ctx}(\mathtt{cc}, \Gamma) = \Gamma_{\mathtt{cc}}$*, and* $\Gamma' = \Gamma'_1, \Gamma'_2$ *such that* $\Gamma = \Gamma_{cc}, \Gamma'_1$*, and* $\mathtt{H} \vDash \sigma(\mathtt{F}_{cc})$*, and* $\sigma' \subset \sigma$*,then* $\mathtt{H} = \mathtt{H}_1 \uplus \mathtt{H}_2$ *such that* $\mathtt{H}_1 \vDash [\![\, \Gamma'_2 \,]\!]_E$*, and* $\mathtt{H}_2 \vDash \sigma(\Delta)$*.*

Lemma 16 (Safety of CC)

If $\Omega; \Gamma; \Pi_1 \vdash \mathtt{cc} : \Pi_2$*, and* $\mathtt{shape}(\mathtt{cc}, E, \Gamma) = \mathtt{F}$*, and* $\mathtt{H}_1 \vDash \mathtt{F}$*, and* $[\![\, \mathtt{cc} \,]\!]_E = (\mathtt{F}_{cc}, \sigma)$*, and* $\sigma \subseteq \sigma'$*, and* $S \subseteq dom(\mathtt{H}_1)$*, and for all* x *such that* $x \in dom(\Pi_1)$*,* $x \in dom(\sigma')$*, for all* $\mathtt{tm}$ *such that* $\Pi_1(\mathtt{tm}) = \mathtt{yes}$*,* $\sigma'(\mathtt{tm}) \in dom(\mathtt{H}_1)$ *or* $\sigma'(\mathtt{tm}) = 0$*, or* $\mathtt{tm} = \$x$*, and* $E(\$x) \in dom(\mathtt{H}_1)$ *or* $E(\$x) = 0$*, then*

- *either* $\mathtt{MP}(\mathtt{H}_1 \uplus \mathtt{H}_2; S; \mathtt{F}; \sigma') = (S', \sigma'')$*, and* $\mathtt{MP}$ *will not access location* l *if* $l \notin dom(\mathtt{H}_1)$*, and* $\forall x \in dom(\Pi_2).\, x \in dom(\sigma'')$*, for all* $\mathtt{tm}$ *such that* $\Pi_2(\mathtt{tm}) = \mathtt{yes}$*,* $\sigma''(\mathtt{tm}) \in dom(\mathtt{H}_1)$ *or* $\sigma''(\mathtt{tm}) = 0$*.*
- *Or* $\mathtt{MP}(\mathtt{H}_1 \uplus \mathtt{H}_2; S; \mathtt{F}; \sigma') = \mathtt{NO}$*, and* $\mathtt{MP}$ *will not access location* l *if* $l \notin dom(\mathtt{H}_1)$*.*

Theorem 17 (Progress and Preservation)

1. *if* $\Omega_\$ \vdash e : t$ *and* $\vdash E_\$: \Gamma_\$$ *then either* $e = n$ *or exists* e' *such that* $(E; e) \longmapsto e'$*, and* $\Omega; \Gamma \vdash e' : t$
2. *if* $\Omega_\$; \Gamma; \Theta; \Delta \vdash \mathtt{stmt} : (\Gamma'; \Theta'; \Delta')$*,* $\vdash E_\$: \Gamma_\$$*,* $\mathtt{H} \vDash [\![\, \Gamma \,]\!]_E \otimes \Delta$ *then either*
   - $\mathtt{stmt} = \mathtt{skip}$ *or* $\mathtt{fail}$ *or*
   - $\mathtt{stmt} = \mathcal{C}_{call}[x := f\ (\ v_1, \cdots, v_n\ )]$ *or*
   - *exists* $\mathtt{stmt}'$*,* E'*,* $\mathtt{H}'$ *such that* $(E; \mathtt{H}; \mathtt{stmt}) \longmapsto (E'; \mathtt{H}'; \mathtt{stmt}')$*, and exists* Γ''*,* Θ''*,* Δ'' *such that* $\Omega_\$; \Gamma''; \Theta''; \Delta'' \vdash \mathtt{stmt}' : (\Gamma'; \Theta'; \Delta')$*, and* $\vdash E'_\$: \Gamma''_\$$*,* $\mathtt{H}' \vDash ([\![\, \Gamma \,]\!]''_{E'}) \otimes \Delta''$*, and for all* $\$s \in dom(\Gamma \cup \Theta)$*,* $(\Gamma \cup \Theta)(\$s) = (\Gamma'' \cup \Theta'')(\$s)$*.*
3. *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* $\vdash (E'; \mathtt{H}'; S'; \mathtt{fb}')\ \mathsf{OK}$