Low-Level Liquid Types *

Patrick Rondon
UCSD
prondon@cs.ucsd.edu

Ming Kawaguchi
UCSD
mwookawa@cs.ucsd.edu

Ranjit Jhala
UCSD
jhala@cs.ucsd.edu

Abstract

We present *Low-Level Liquid Types*, a refinement type system for C based on *Liquid Types*. Low-Level Liquid Types combine refinement types with three key elements to automate verification of critical safety properties of low-level programs: First, by associating refinement types with individual heap locations and precisely tracking the locations referenced by pointers, our system is able to reason about complex invariants of in-memory data structures and sophisticated uses of pointer arithmetic. Second, by adding constructs which allow strong updates to the types of heap locations, even in the presence of aliasing, our system is able to verify properties of in-memory data structures in spite of temporary invariant violations. By using this strong update mechanism, our system is able to verify the correct initialization of newly-allocated regions of memory. Third, by using the abstract interpretation framework of Liquid Types, we are able to use refinement type inference to automatically verify important safety properties without imposing an onerous annotation burden. We have implemented our approach in CSOLVE, a tool for Low-Level Liquid Type inference for C programs. We demonstrate through several examples that CSOLVE is able to precisely infer complex invariants required to verify important safety properties, like the absence of array bounds violations and null-dereferences, with a minimal annotation overhead.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Languages, Reliability, Verification

Keywords Liquid Types, Type Inference, Dependent Types, C

1. Introduction

Static verification is a crucial last line of defense at the lowest levels of the software stack, as at those levels we cannot fall back on dynamic mechanisms to protect against bugs, crashes, or malicious attacks. Recent years have seen significant progress on *automatic* static verification tools for systems software. These tools employ abstract interpretation [5, 18] or software model checking [3, 17, 7, 33] to infer path-sensitive invariants over *program variables* like status flags and counters and thereby verify control-sensitive safety properties. Unfortunately, these approaches have been proven insufficient for verifying data-sensitive properties of values stored in lists, trees, *etc.*, as this requires the precise inference of invariants of data values stored within *unbounded collections* of heap-allocated cells.

In previous work we introduced *Liquid Types* [29], a refinement type system for ML that marries the ability of ML types to infer coarse invariants for polymorphic data structures (and higher-order functions) with the ability of predicate abstraction and SMT solvers to infer path-sensitive invariants of individual variables. We demonstrated that this symbiotic combination enables the highly automated verification of complex data-sensitive properties of high-level, functional programs [20]. Unfortunately, the very nature of low-level, imperative code, typically written in C, makes the translation of type-based mechanisms to the setting of systems software verification extremely challenging.

Lack of Types First, due to the presence of casts and pointer arithmetic, low-level systems code is essentially *untyped*. C's type system is designed only to allow the compiler to determine the number of bytes that should be read or written by each instruction, and hence, unlike the type systems of higher-level languages, C's types provide no invariants about data values.

Mutation Second, mutation makes the very notion of type refinement problematic. The key idea in refinement types is to adorn the basic underlying types with *refinement predicates* over program variables. For example, in an ML program, the refinement type $\{\nu\!:\!\texttt{int} \mid \texttt{x} \leq \nu\}$ describes an integer that is greater than the program variable x. However, this type is meaningless if the value of x can change over time.

Unbounded Collections Third, even if we could meaningfully track mutation, we cannot always uniquely identify the object being mutated. In particular, the presence of unbounded collections means that we must represent many elements of a collection by a single type. This makes it impossible to strongly update the type of an element in the collection, creating a major loss of precision in the presence of the temporary invariant violations common in low-level programs.

We introduce *Low-Level Liquid Types* (LTLL), a static refinement type system for C that enables the precise verification and inference of data-sensitive properties of low-level software. LTLL tackles the above challenges via a three-tiered design.

First, LTLL is founded on a new Basic type system that classifies values and heaps. A value is either a *datum* of a given size , *e.g.,* a 4-byte integer or a 1-byte character, or a *reference* corresponding to a pair of a heap *location* and an *offset* within the location. Intuitively, an offset corresponds to a field (resp. cell) of the structure (resp. array) resident at the location. A heap is a map from locations to a sequence of offset-value bindings that define the contents of

* This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and a gift from Microsoft Research.

POPL'10, January 17–23, 2010, Madrid, Spain.

the given location. By precisely tracking arithmetic on offsets, Basic types provide coarse invariants about the basic shapes of data values.

Second, each Basic type is refined with a predicate that captures precise properties of the values defined by the type. LTLL makes a clear separation between immutable state, which is tracked using a traditional type environment, and mutable state, which is tracked in a flow-sensitive heap. We ensure soundness by restricting the refinements to *pure* predicates that refer only to immutable values. Of course, in C all entities are mutable. We recover precision for stack-allocated variables by first carrying out an SSA renaming, which creates different (immutable) versions for the variables at different program points.

Third, we recover precision for heap-allocated locations by using the Basic type information to strongly update the types of the heap's contents on writes through pointers. Since such strong updates are unsound when several physical heap locations are represented by a single type, LTLL distinguishes between *abstract* locations which summarize a collection of physical memory locations and *concrete* locations which describe exactly one physical memory location. LTLL enables strong updates by enforcing the requirement that all pointer reads and writes are to concrete locations, and by employing two mechanisms, inspired by version control systems, to account for aliasing: *unfold*, which "checks out" a concrete reference to a particular location from the set described by an abstract location, and its dual, *fold*, which "commits" the changes made to the particular location back into the abstract location after ensuring that the particular location satisfies the invariants of the abstract location. Together, the automatically-inserted fold and unfold annotations ensure that the invariants for an abstract location soundly apply to *all* the elements that correspond to that location, while simultaneously allowing strong updates. This is crucial, as strong updates are essential for both establishing and tolerating temporary violations of the invariants that are ubiquitous in low-level code.

Finally, LTLL uses the abstract interpretation framework of Liquid Types to permit automatic inference of the refinements. The typing rules directly correspond to an algorithm that generates a system of subtyping constraints over templates containing variables that stand for the unknown refinements. These constraints reduce to a system of logical implication constraints that are solved via predicate abstraction in order to yield the refinement types, and hence precise invariants, for different program elements.

To demonstrate the utility of LTLL, we have implemented it in CSOLVE, a prototype static verifier for C. CSOLVE takes as input a C program and a set of logical predicates and returns as output the inferred dependent types of local variables and heap contents along with a report of any type errors that occurred. Through a set of challenging case studies, we show how the combination of types and predicate abstraction enables the precise, path-sensitive verification and inference of control-sensitive properties of individual variables *and* data-sensitive properties of aggregate structures.

2. Overview

We start with a high-level overview of Low-Level Liquid Types, and then, via a sequence of examples, we illustrate how they enable the precise static verification and inference of program invariants in the presence of challenging low-level programming constructs, including pointer arithmetic, memory allocation, temporary invariant violations, aliasing and data structures.

Basic Types Our system is based on a new Basic type system for C where every program variable is either a *basic data value* of some size, *e.g.,* a 4-byte integer denoted by `int`, or a *reference* comprising a *location* and an *index* within the location denoted by $\mathtt{ref}(\ell, i)$, where ℓ is the location and i the index within the location. An index is either a natural number n, which is a singleton offset used to model pointers to specific fields of a structure, or of the form n^{+m}, which is a sequence of offsets $\{n + lm\}_{l=0}^{\infty}$ used to model pointers into an array of items of size m that starts at offset n. Thus, $\mathtt{ref}(\ell, 4)$ is a (possibly null) pointer that refers to a location ℓ at (field) offset 4, while $\mathtt{ref}(\ell, 0^{+4})$ is a (possibly null) pointer that refers to a location within an array of 4-byte integers.

Basic Heaps To ensure the soundness of types in the presence of mutation, our representation of program state is partitioned into an *environment*, which is a standard sequence of type bindings for *immutable* variables, and a *heap*, which is a mapping from locations ℓ to a set of index-type pairs that describe the contents of the location, called a *block*. For example, the heap

$$\begin{aligned}\ell^1 &\mapsto 0{:}\mathtt{int}, 4{:}\mathtt{int}\\ \ell^2 &\mapsto 0^{+1}{:}\mathtt{char}\end{aligned}$$

has two locations. The first, ℓ^1, contains a structure with two integer fields (at offsets 0 and 4 respectively). The second, ℓ^2, contains an array of one-byte characters (denoted `char`).

Refinement Types and Heaps In our system, program invariants are captured via *refinement types* [25, 14, 4, 29] denoted by $\{\nu{:}\tau \mid e\}$ where τ is the Basic type being refined, ν is a special *value variable* that denotes the value being described, and e is the *refinement predicate*, a Boolean-valued expression containing the value variable. Intuitively, the refinement type describes the set of values c of the Basic type τ such that the predicate $e[c/\nu]$ evaluates to true. Thus, $\{\nu{:}\mathtt{int} \mid 0 \leq \nu\}$ describes the set of non-negative integers, and $\{\nu{:}\mathtt{ref}(\ell, 0) \mid \nu \neq 0\}$ describes the set of non-null references to a location ℓ at offset 0. A *refinement heap* is a heap where each location is mapped to a sequence of offset-refinement-type pairs. For example, $\ell_1 \mapsto 0{:}\{\nu{:}\mathtt{int} \mid 0 \leq \nu\}$ is a heap with a location ℓ_1 which contains a non-negative integer at offset 0.

Liquid Types A *logical qualifier* is a Boolean-valued expression over the program variables, the value variable ν, and a placeholder variable $\star$. We say that a qualifier q matches the qualifier q' if replacing some subset of the free variables in q with $\star$ yields q'. For example, the qualifier $\nu \leq x + y$ matches the qualifier $\nu \leq \star + \star$. We write $\mathbb{Q}^\star$ for the set of all qualifiers not containing $\star$ that match some qualifier in $\mathbb{Q}$. In the rest of this section, let $\mathbb{Q}$ be the set

$$\begin{aligned}\{&0 \leq \nu, \nu = \star + \star, \nu = BS(\nu),\\ &BS(\nu) = BS(\star), BE(\nu) = BS(\nu) + \star\}\end{aligned}$$

The terms $BS(\cdot)$ and $BE(\cdot)$ are uninterpreted function applications denoting the start and end addresses of memory blocks; we will explain these shortly. A *liquid type* over $\mathbb{Q}$ (abbreviated to just liquid type) is a refinement type where the refinement predicates are *conjunctions* of qualifiers from $\mathbb{Q}^\star$. Our system enables inference by requiring that the certain entities, *e.g.,* loop-modified variables, functions and blocks in aggregate structures, have liquid types.

2.1 Local Invariants

We begin by showing how our system uses *local* refinements for individual program variables to verify the safety of the pointer dereferences in the `make_string` function shown in Figure 1. The function takes an integer parameter `n`, allocates a new block of memory of size `n`, iterates over the block using `str` to initialize it, and returns a reference, `res`, to the block.

Basic Types First, we describe the Basic types computed for each variable. The function calls `malloc` to create a new heap location ℓ^1 and returns a pointer to the location with offset 0. Thus, `res` gets the Basic type $\mathtt{ref}(\ell^1, 0)$. `str` is initialized with `res` but is updated inside the loop with an increment of 1. Hence, it gets assigned the Basic type $\mathtt{ref}(\ell^1, 0^{+1})$. The loop index `i` gets the Basic type `int`.

```
char *make_string(int n) {
   char *res;
   char *str;
1: if (n < 0) return NULL;
2: res = (char *)malloc(n*sizeof(char));
3: str = res;
4: for(int i = 0; i < n; i++) {
5:   *str++ = '\0';
   }
6: return res;
}
```

Figure 1. Example: `make_string`

```
typedef struct {
  int len;
  char *str;
} string;

string *new_string(int n, char c){
   string *s;
   char   *str;
0: if (n < 0) return NULL;
1: s = (string *)malloc(sizeof(string));
2: s->len = n;
3: str = make_string(n);
4: s->str = str;
5: init_string(s,c);
   return s;
}

void init_string(string *s, char c){
  for (int i = 0; i < s->len; i++) {
    s->str[i] = c;
  }
}
```

Figure 2. Example: `new_string`

```
typedef struct _slist {
  struct _slist *next;
  string        *s;
} slist;

slist *new_strings(int n) {
   string *s;
   slist *sl, *t;
1: sl = NULL;
2: for (int i = 1; i < n; i++) {
3:   s = (string *)malloc(sizeof(string));
4:   s->len = i;
5:   s->str = make_string(i);

6:   t = (slist *)malloc(sizeof(slist));
7:   t->s = s
8:   t->next = sl;
9:   sl = t;
  }

  return sl;
}
```

Figure 3. Example: `new_strings`

Pointer Allocation and Arithmetic To specify when it is safe to dereference a pointer, we refine the output type of `malloc` so that it contains information about the size of the allocated block. In particular, in our system `malloc` returns a value of type

$$\{\nu:\mathtt{ref}(\ell,0) \mid \mathit{BLen}(\nu,\mathtt{n})\}$$

where `n` is the size argument passed to `malloc` and BLen is the following *block length predicate*:

$$\mathit{BLen}(\nu,n) \doteq BS(\nu)=\nu \wedge BE(\nu)=\nu+n$$

The refinement states that the return value is equal to the start of the location it points to ($BS(\nu)$), and that the end of the allocated region ($BE(\nu)$) is `n` bytes from the beginning. We adopt a *logical* model of memory where allocated blocks are considered to be infinitely far apart. We reflect this in our type system by refining the output types of pointer arithmetic operations to stipulate that when a pointer `x` is incremented by a value `i` the result has refinement

$$\mathit{PAdd}(\nu,\mathtt{x},\mathtt{i}) \doteq \nu=\mathtt{x}+\mathtt{i} \wedge BS(\nu)=BS(\mathtt{x}) \wedge BE(\nu)=BE(\mathtt{x})$$

which states that the result is an appropriately offset pointer into the *same* block. Finally, to specify the safety of pointer dereferences, we stipulate that whenever a pointer x is dereferenced for reading or writing, it has the *bounds-safe* type

$$\{\nu:\mathtt{ref}(\ell,0^{+1}) \mid BS(\nu)\leq\nu \wedge \nu<BE(\nu)\}$$

Safety Verification To verify that the pointer dereference on line 5 is safe, we must verify that `str` has the bounds-safe type; this will require determining that $\mathtt{str}=\mathtt{res}+\mathtt{i}$. This is challenging for a type system, as both `str` and `i` are mutated by the loop. Our system addresses this problem by using SSA renaming to compute different types for the different versions of mutated variables. In the sequel, let $\mathtt{x}_\mathtt{j}$ be the SSA name of `x` at line `j`. Thus, from the `malloc` at line 2 our system deduces that $\mathtt{res}_2$ has type

$$\{\nu:\mathtt{ref}(\ell^1,0) \mid \mathit{BLen}(\nu,\mathtt{n})\} \tag{1}$$

i.e., that `res` is a pointer to the start of a new location ℓ^1 whose size is `n` bytes. This same type is assigned to $\mathtt{str}_3$. Next, our system uses the qualifiers $\mathbb{Q}$ and an SMT solver to infer that at line 5 $\mathtt{i}_5$ and $\mathtt{str}_5$ have the respective types

$$\{\nu:\mathtt{int} \mid 0\leq\nu<\mathtt{n}\}$$
$$\{\nu:\mathtt{ref}(\ell^1,0^{+1}) \mid \mathit{PAdd}(\nu,\mathtt{res}_2,\mathtt{i}_5)\}$$

Notice that these types are loop invariants. They hold the first time around the loop as initially `i` is 0 and `str` is equal to `res`. The types are inductive invariants, as each loop iteration increments `i` and `res`. Thus, our system uses an SMT solver to combine the above facts with (1) and deduce that at line 5 $BS(\mathtt{str}_5)\leq\mathtt{str}_5 \wedge \mathtt{str}_5<BE(\mathtt{str}_5)$, *i.e.,* that $\mathtt{str}_5$ has the bounds-safe type and hence the pointer dereferences at line 5 of `make_string` are safe.

Function Types Finally, note that `make_string` returns the pointer `res` (*i.e.,* $\mathtt{res}_2$) on line 6. Thus, using the type from (1) and the fact that the location ℓ^1 was freshly generated via `malloc` , our system concludes that `make_string` has the type:

$$\forall\ell^1.(\mathtt{n}:\mathtt{int})/\mathtt{emp} \rightarrow \{\nu:\mathtt{ref}(\ell^1,0) \mid \mathit{BLen}(\nu,\mathtt{n})\}/\ell^1 \mapsto 0^{+1}:\mathtt{char} \tag{2}$$

That is, the function takes an integer `n` and an empty heap (*i.e.,* does not touch any pre-existing heap cells) and returns a pointer to the start of a new `char` array of size `n`.

2.2 Heap-block Invariants

Next, we show how our system uses refinements to verify safety properties of blocks of data residing in the heap. Consider the `new_string` function shown in Figure 2. This function takes a parameter, `n`, and produces a `string` structure encoding a string of length `n`. The `string` structure has two fields: `len`, the length of the string, and `str`, a pointer to the contents of the string. The programmer intends that the fields obey the following two invariants:

(I_1) the `len` field is non-negative, and

(I_2) the `str` field points to a `char` array of size `len`.

Note that these invariants do not hold at all points during the lifetime of the structure; instead, the programmer establishes them on lines 1-4, and then calls the procedure `init_string` that fills in the string with the supplied character `c`.

Next, we show how our system precisely tracks updates to the structure, tolerating the early stages in which the invariant does not hold, in order to verify the safety of the pointer dereferences within `init_string`.

First, the `malloc` in line 1 creates a new location on the heap, ℓ^2, and gives `s` the type $\mathtt{ref}(\ell^2,0)$, stating that it points into this location at offset 0. Initially, this location contains an 8-byte block (the size of the string structure), and so at line 2 the heap is

$\ell^2 \mapsto$ uninitialized 8-byte block

In line 2, we assign n to the len field of s, which creates a new binding in the heap for ℓ^2 at the offset corresponding to the field len, namely 0, since len is the first element of the structure. Thus, at line 3 the heap is

$\ell^2 \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid \nu = \texttt{n}\}$, uninitialized 4-byte block

Next, in line 3, the call to make_string creates a new location and assigns to str a pointer to the new location, with the type shown in 2 (and 1). Thus, at line 4 the heap contains two locations

$\ell^1 \mapsto 0^{+1}\!:\!\texttt{char}$

$\ell^2 \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid \nu = \texttt{n}\}$, uninitialized 4-byte block

In line 4, the value of str is assigned to s->str, which creates a binding at the corresponding offset in ℓ^2, namely 4, as the first field, len, was an int which is 4 bytes long. Thus, at line 5 the heap is

$\ell^1 \mapsto 0^{+1}\!:\!\texttt{char}$

$\ell^2 \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid \nu = \texttt{n}\}, 4\!:\!\{\nu\!:\!\texttt{ref}(\ell^1, 0) \mid \nu = \texttt{str}\}$

Finally, at line 5 we have the call to init_string. At the callsite, our system uses the qualifiers in $\mathbb{Q}$, and the type of str to infer that the previously shown heap binding for ℓ^2 is subsumed by

$\ell^2 \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid \nu = \texttt{n}\}, 4\!:\!\{\nu\!:\!\texttt{ref}(\ell^1, 0) \mid BLen(\nu, \texttt{n})\}$

As the value at offset 0 equals n, the above block is subsumed by

$\ell^2 \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid \nu = \texttt{n}\}, 4\!:\!\{\nu\!:\!\texttt{ref}(\ell^1, 0) \mid BLen(\nu, @0)\}$

where n is replaced by @0, a name that denotes the value within the same block at offset 0. Finally, our system uses the test at line 0 to deduce that n is non-negative at the callsite, so init_string is called with the heap h defined as

$$h \doteq \ell^2 \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid 0 \leq \nu\}, 4\!:\!\{\nu\!:\!\texttt{ref}(\ell^1, 0) \mid BLen(\nu, @0)\},$$
$$\ell^1 \mapsto 0^{+1}\!:\!\texttt{char}$$

Note that, as the len field of a string structure is located at offset 0 and its str field is located at offset 4, the bindings for ℓ^2 capture exactly the structure invariants $\mathsf{I}_1, \mathsf{I}_2$ intended by the programmer. Moreover, even though the invariants don't hold everywhere, our system is able to use strong updates to establish them at function call boundaries. Thus, our system infers that the function init_string has the type

$$\forall \ell^1, \ell^2.(\texttt{s}\!:\!\texttt{ref}(\ell^2, 0))/h \rightarrow \texttt{void}/h$$

and, via reasoning analogous to that for make_string, our system verifies the safety of array accesses in init_string.

2.3 Data Structure Invariants

In new_string, s pointed to exactly one heap location, ℓ^1, throughout the execution of the function. Thus, we could soundly perform strong updates to the block describing the contents of ℓ^1; this allowed us to determine that the strings built by new_string satisfied the desired invariants. Unfortunately, we cannot soundly use strong updates when dealing with *collections* of locations.

Consider the function new_strings shown in Figure 3. This function takes an integer parameter, n, and creates a list of strings of lengths from 1 to n, all of which satisfy the invariants $\mathsf{I}_1, \mathsf{I}_2$. This is accomplished by looping from 1 to n, allocating memory for a new string and assigning the pointer to this memory to s (line 3), initializing it as in new_string (lines 4,5), and inserting s into a list of strings (lines 6,7,8).

Note that s points to *many* different concrete locations over the course of executing the function; this is in contrast to the previous functions, in which pointers only pointed to a *single* concrete location while the function was executed. We formalize this distinction by saying that s points to an *abstract location* $\tilde{\ell}$. That is, in our system, s has the Basic type $\texttt{ref}(\tilde{\ell}, 0)$, which states that it refers to the offset 0 within (one of) *many* possible locations.

Observe that it is not sound to perform strong updates to an abstract location's type. To see why, suppose that we had strongly updated $\tilde{\ell}$ as we did when analyzing new_string. Then we would assign $\tilde{\ell}$ a block type as follows:

$$\tilde{\ell} \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid \nu = \texttt{i}\}, \ldots$$

There are two problems with this type. First, every string has a different length, and yet we only assign a single length for all strings. Second, at the end of the function, i has the value n, while none of the strings in the list has length n! Thus, while we need strong updates to establish the desired invariants for each string, we clearly cannot soundly perform strong updates on the types of abstract locations.

We solve this problem with the following crucial observation. Suppose that the code uses a pointer s to access a collection of locations $\tilde{\ell}$. As long as we do not modify s or use other pointers to $\tilde{\ell}$, only one *particular* concrete location from the set represented by $\tilde{\ell}$ can be modified at a time. Thus, when a pointer to $\tilde{\ell}$ is *first* used, we can *unfold* the abstract location into a fresh concrete location, ℓ_j, which inherits $\tilde{\ell}$'s invariant. As long as $\tilde{\ell}$ is only accessed by a pointer to ℓ_j, we can soundly perform strong updates on ℓ_j's type. However, as soon as another pointer to $\tilde{\ell}$ is used, the possibility of aliasing means we can no longer rely on ℓ_j's type to be accurate. Thus, *before* we access an abstract location via another pointer of type $\tilde{\ell}$, we *fold* the concrete location ℓ_j back into the collection by verifying that ℓ_j satisfies $\tilde{\ell}$'s invariants and removing it from the heap. The other pointer then gets its own unfolded copy of the location, and can strongly update it, until it gets folded back into the collection, and so on. Our system automatically places folds and unfolds in the code in a manner that ensures that: (1) every heap access occurs via a reference to a concrete location, (2) every abstract location has at most one corresponding concrete location in the heap at any point in time. In this way, our system can soundly establish invariants about unbounded data structures in spite of temporary invariant violation even in the presence of aliasing.

We now illustrate the above mechanism using the code in Figure 3. We will say that, within the body of the loop, s points to some concrete location, ℓ_j, which is an instance of $\tilde{\ell}$. We will use strong updates, as in the previous examples, to verify that ℓ_j has the desired invariants, *i.e.,* that

$$\ell_j \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid 0 \leq \nu\}, 4\!:\!\{\nu\!:\!\texttt{ref}(\ell_2, 0) \mid BLen(\nu, @0)\}.$$

Finally, at the end of the loop — *i.e.,* before we access another pointer into $\tilde{\ell}$ in the next iteration — we *fold* the concrete location ℓ_j into the collection by ensuring that it satisfies $\tilde{\ell}$'s invariants, *i.e.,* by stipulating that at the end of of the loop, the block ℓ_j is a subtype of the block $\tilde{\ell}$. In this manner, our system performs strong updates *locally* and infers using $\mathbb{Q}$ that at the end of the new_strings, the heap is of the form

$\tilde{\ell} \mapsto 0\!:\!\texttt{ref}(\tilde{\ell}, 0), 4\!:\!\texttt{ref}(\tilde{\ell^1}, 0)$

$\tilde{\ell^1} \mapsto 0\!:\!\{\nu\!:\!\texttt{int} \mid 0 \leq \nu\}, 4\!:\!\{\nu\!:\!\texttt{ref}(\tilde{\ell^2}, 0) \mid BLen(\nu, @0)\}$

$\tilde{\ell^2} \mapsto 0^{+1}\!:\!\texttt{char}$

a	$::=$		**Pure Expressions**
	$\mid$	$\langle w \rangle_n$	integer constant
	$\mid$	$\mathtt{ref}(r, n)$	reference constant
	$\mid$	x	variable
	$\mid$	$@i$	block offset
	$\mid$	$a_1 + a_2$	integer arithmetic
	$\mid$	$a_1 +_p a_2$	pointer arithmetic
	$\mid$	$a_1 \sim a_2$	pointer comparison
	$\mid$	$\mathtt{assert}(a)$	assertion
	$\mid$	$F(a \ldots)$	unint. application
e	$::=$		**Expressions**
	$\mid$	a	pure expression
	$\mid$	$\mathtt{if}\ a\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$	if-then-else
	$\mid$	$\mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2$	binding
	$\mid$	$\mathtt{letu}\ x\ =\ [\mathtt{unfold}\ \ell]\ a\ \mathtt{in}\ e$	location unfold
	$\mid$	$[\mathtt{fold}\ \ell]$	location fold
	$\mid$	$*a$	pointer read
	$\mid$	$*a_1 := a_2$	pointer write
	$\mid$	$\mathtt{malloc}(\ell, a)$	allocation
	$\mid$	$[\ell \ldots]\ \mathtt{f}(a \ldots)$	function call
f	$::=$		**Functions**
	$\mid$	$\mathtt{fun}(x \ldots)\{e\} : S$	definition
p	$::=$		**Programs**
	$\mid$	$\mathtt{letf}\ \mathtt{f}\ =\ f\ \mathtt{in}\ p$	function binding
	$\mid$	$\mathtt{f}()$	main call

Figure 4. NanoC syntax

ℓ	$::=$		**Locations**
	$\mid$	$\tilde{\ell}$	abstract location
	$\mid$	ℓ_j	concrete location
i	$::=$		**Indices**
	$\mid$	n	singleton
	$\mid$	n^{+m}	lower-bounded sequence
τ	$::=$		**Basic Types**
	$\mid$	$\langle n \rangle_i$	integer
	$\mid$	$\mathtt{ref}(\ell, i)$	reference
T	$::=$		**Refined Types**
	$\mid$	$\{\nu : \tau \mid a\}$	refined Basic type
b	$::=$		**Blocks**
	$\mid$	$i : T \ldots$	block
h	$::=$		**Heaps**
	$\mid$	$\mathtt{emp}$	empty heap
	$\mid$	$h * \ell \mapsto b$	location binding
S	$::=$		**Function Schemas**
	$\mid$	$(x : T \ldots)/h \rightarrow T/h$	function type
	$\mid$	$\forall \ell . S$	location quantification

Figure 5. NanoC types

Thus, our system infers that the function returns a list ($\tilde{\ell}$) of pointers to `string` structures (ℓ^1) each of which satisfy invariants I_1 and I_2.

Plan This concludes a high-level overview of LTLL. Next we formalize our core language (Section 3) and static type system and state the type soundness theorem (Section 4). Next, we describe our experimental evaluation via a set of challenging case studies (Section 5), We then survey the diverse lines of research to which LTLL is related (Section 6) before concluding (Section 7).

3. Language

In this section, we present the syntax and types of NanoC, a simple C-like language with integers and pointers.

3.1 Syntax

The syntax of NanoC is shown in Figure 4. We give an overview of the language's features below.

Pure Expressions We distinguish the *pure* expressions of NanoC, which do not access the heap, from its potentially *impure* expressions. The pure expressions of NanoC, denoted by a, include integer constants, variables, integer and pointer arithmetic, integer and pointer comparisons, and assertions, which allow the static checking of arbitrary predicates. NanoC uses the C convention that nonzero values represent truth and all other values represent falsehood. Thus, the generic arithmetic operator, denoted by $+$, includes comparisons and boolean operations. Uninterpreted applications and block offsets do not appear in programs; they are used solely in the refinements discussed in Section 3.2. Similarly, reference constants pointing to run-time heap locations r do not appear in source programs, but are the result of evaluating reference-typed expressions. Note that pure expressions are guaranteed to evaluate to a value.

Expressions The impure expressions of NanoC, denoted by e, include the pure expressions, as well as if-then-else expressions, let bindings, reads from and writes to memory, memory allocation, location folding and unfolding, and function calls. Note that all bindings are to immutable variables — all mutation is factored into the heap. Next, we examine location unfolding and function calls in more detail.

Location Fold and Unfold Our goal is to verify invariants which hold on in-memory data structures. These invariants are represented as types attached to *abstract* heap locations, each of which may represent several *concrete* (actual, run-time) heap locations. Verifying properties of the data at these abstract locations in the presence of temporary invariant violation would seem to require performing strong updates on the types of abstract locations. Unfortunately, this would be unsound, since a single abstract location can represent several concrete locations.

However, at run-time a reference will only point to a single concrete location at a time. Thus, operations on abstract locations through a single reference will only affect a single concrete location. Intuitively, if we can get access to this concrete location, we can soundly perform strong updates on it.

Our intuition follows a version control metaphor. Before using a pointer, we can "check out a copy" of its abstract location, giving a concrete location for the pointer which has the same type as the abstract location — a "working copy". As long as the abstract location is accessed only through this pointer to the working copy, it will be sound to perform strong updates on the type of the new concrete location. Finally, if it becomes necessary to use another pointer to the same abstract location, we "check in" the concrete location by checking that it satisfies the same invariant as the corresponding abstract location. The concrete location is then discarded so that no further modification can be made to the working copy.

The "check out" operation is implemented via the $\mathtt{letu}\ x\ =\ [\mathtt{unfold}\ \ell]\ a\ \mathtt{in}\ e$ construct, where a is a reference to abstract location $\tilde{\ell}$. The expression creates a new concrete location corresponding to $\tilde{\ell}$; a reference to this new location is bound to x in e. The "check in" operation is implemented via the $[\texttt{fold}\ \ell]$ expression, which verifies that the concrete location corresponding to $\tilde{\ell}$ satisfies the same invariant as $\tilde{\ell}$. These procedures and the distinction between abstract and concrete locations are discussed in more detail in the context of their static typing rules in Section 4.1.

Function Calls Since functions take reference parameters, they can operate on arbitrary memory locations. Thus, we allow function types to be quantified over the locations they operate on and augment the function call expression with syntax for instantiating the quantified locations: the expression $[\ell \ldots]\ f(x \ldots)$ calls function f with parameters $x \ldots$, instantiating the location variables in the type schema of f with locations $\ell \ldots$.

Programs A NanoC program, denoted by p, is a sequence of function definitions followed by a call to one of the previously-defined functions. The program is evaluated by evaluating the function call.

3.2 Types

The types of NanoC are shown in Figure 5. NanoC has a system of refined base types, T, dependent heaps, h, and dependent function schemas, S.

Locations and References The NanoC *locations*, ℓ, denote areas of the heap. We use $\tilde{\ell}$ to denote an *abstract location*; abstract locations cannot be read from or written to. We use ℓ_j to denote a *concrete location*; only concrete locations can be read from or written to. Every concrete location ℓ_j (resp. ℓ^i_j) corresponds to some abstract location $\tilde{\ell}$ (resp. $\tilde{\ell^i}$), and we require for soundness that there is at most one concrete location corresponding to a particular abstract location at any given program point. We call references to abstract locations *abstract references* and references to concrete locations *concrete references*.

Indices The integer and reference types of NanoC make use of *indices*, i, which are a shorthand notation for single integers and arithmetic sequences. The index n represents the singleton offset set $\{n\}$; the index n^{+m} represents the sequence of offsets $\{n + lm\}_{l=0}^{\infty}$. We write i^+ to refer to an index which represents a sequence.

Basic Types The base types, τ, of NanoC include integer and reference types. We use $\langle n \rangle_i$ to denote the type of n-byte integers x such that $x \in i$. We use $\texttt{ref}(\ell, i)$ to denote the type of references to location ℓ at an offset $x \in i$ within that location.

Refined Types A refined type, T, has the form $\{\nu : \tau \mid a\}$, where τ is a Basic type and a is a pure expression called a *refinement predicate*. Note that we can directly embed refinement predicates as quantifier free formulas in the (decidable) theory of equality, linear arithmetic and uninterpreted functions (EUFA). Intuitively, the type $\{\nu : \tau \mid a\}$ denotes values v of Basic type τ such that $a[v/\nu]$ evaluates to true. We use the following type abbreviations: `int` abbreviates $\langle W \rangle_{-\infty^{+1}}$, where W is the length in bytes of a word, `char` abbreviates $\langle 1 \rangle_{-\infty^{+1}}$, and `void` abbreviates $\langle 0 \rangle_0$. When it is unambiguous from the context, we use τ to abbreviate the type $\{\nu : \tau \mid true\}$. Similarly, when the Basic type τ is clear from the context, we use $\{a\}$ to abbreviate $\{\nu : \tau \mid a\}$.

Blocks A block, b, represents the contents of a heap location. The types of the block's contents at various offsets are given by bindings $i : T$ which state that the values at the offset(s) i have the type T. Within a block, no two index bindings may overlap.

Heaps A heap type, h, represents the contents of the run-time heap, giving a block type to each location in the heap. The contents of heap location ℓ are given by a binding to a block b, written $\ell \mapsto b$. We can form the concatenation of two heaps h_1 and h_2 as $h_1 * h_2$; the resulting heap contains all bindings present in either h_1 or h_2. Our heaps enjoy the following properties: (1) no location may be bound twice in a heap, (2) every abstract location in the heap has at most one corresponding concrete location in the heap, and (3) every concrete location in the heap has exactly one corresponding abstract location in the heap. We say that a run-time heap *satisfies* a heap type if every value in the heap has the type specified by the corresponding heap type binding.

Function Schemas We combine refined base types and heap types to form dependent function types and schemas S. A dependent function type consists of an input and output portion. The input portion of a dependent function is a pair $(x_i : T_i \ldots)/h$ of a dependent tuple giving the parameter types and the input heap, *i.e.,* the heap contents required to call the function. The output portion of a dependent function is a pair T/h, called a *world*, containing the return type of the function and the output heap, *i.e.,* the heap contents after the function returns. The types in the output world of a dependent function type may refer to variables bound in the input tuple.

Since functions can take reference parameters, they may operate on arbitrary heap locations. Thus, we allow function schemas to be formed by quantifying a function type over the heap locations it contains.

4. Type System

In this section, we present the typing rules of NanoC, outline a proof of their soundness, and give an overview of how our system enables inference.

4.1 Typing Rules

We begin with a description of NanoC's type environments, rules for type well-formedness, and subtyping. We then discuss several of the most interesting typing rules.

Environments Our typing rules make use of two types of environments: *local environments* and *global environments*. A local environment, Γ, is a sequence of *type bindings* $x : T$ and *guard predicates* e. The former are standard; guard predicates capture the results of conditional guards under which an expression is evaluated. A global environment, Φ, is a sequence of bindings $f : S$ mapping functions to their type schemas.

We assume that suitable renaming has been performed so that no name is bound twice in an environment. An environment is well-formed if each bound type is well-formed in the prefix of the environment that precedes the binding.

$$\Gamma \quad ::= \quad \epsilon \mid x : T; \Gamma \mid a; \Gamma \qquad \text{(Local Environment)}$$
$$\Phi \quad ::= \quad \epsilon \mid \mathtt{f} : S; \Phi \qquad \text{(Global Environment)}$$

Well-Formedness Judgments The judgments of Figure 6 ensure that types, heaps, and worlds are *well-formed* in local environments Γ and heaps h. Intuitively, a type is well-formed in a local environment Γ if its refinement predicate a is a Boolean formula in Γ, written $\Gamma \vdash e$. Additionally, we require that reference types point to heap locations present in h and integer types have non-negative size.

A block is well-formed if no two index bindings overlap and each type is well-formed with respect to the local environment and preceding indices. We distinguish between *concrete blocks*, bound to concrete heap locations, which must have (pure) refinements over immutable variables bound in the environment, and *abstract blocks*, bound to abstract heap locations, which have refinements which may additionally use offset names (*e.g.,* @0) to refer to values at other offsets within the block. We disallow offset names in the refinements for concrete blocks for two reasons. First, they are unnecessary, as we can use names bound in the environment to precisely describe a particular location. Second, they are problematic, as the values at the offsets can be changed by strong updates, thus invalidating the refinements. To ensure that no elements overlap, the block well-formedness rules make use of an auxiliary function, $Ind(i, T)$, which produces the set of offsets occupied by a value of type T located at offsets $x \in i$. For example, $Ind(0, \langle 4 \rangle_0)$ produces the set $\{0, 1, 2, 3\}$, *i.e.,* the set of offsets occupied by a 4-byte integer located at offset 0.

A heap is well-formed if each block is well-formed, no location is bound twice, each abstract location has at most one corresponding concrete location, and each concrete location has a corresponding abstract location. Note that we check blocks bound to abstract locations using abstract block well-formedness and blocks bound to concrete locations using concrete block well-formedness.

A schema is well-formed if all parameters are well-formed with respect to the previous parameters and the input heap, the input heap is well-formed with respect to the parameters, and the output world is also well-formed with respect to the parameters.

Subtyping Judgments The subtyping judgments of NanoC are shown in Figure 8. The rules use set-theoretic inclusion checks between arithmetic sequences represented by indices and logical implication checks over the refinement predicates. To ensure decidability, we embed the implication checks into a decidable logic of Equality, Linear Arithmetic and Uninterpreted Functions (EUFA). We write $[\![a]\!]$ for the embedding of a pure expression a into EUFA. We lift the embedding to environments as follows:

$$[\![x : \{\nu : \tau \mid a\}; \Gamma]\!] \doteq [\![a[x/\nu]]\!] \wedge [\![\Gamma]\!]$$
$$[\![a; \Gamma]\!] \doteq [\![a]\!] \wedge [\![\Gamma]\!]$$
$$[\![\epsilon]\!] \doteq \mathit{true}$$

Most of the rules in Figure 8 are straightforward. Rule [<:-NULLPTR] is used to coerce the integer value 0 into an arbitrary pointer type, allowing the use of null pointers. Rule [<:-ABSTRACT] allows a concrete pointer to be treated as abstract.

Covariant Heap Subtyping Our use of the covariant heap subtyping rule [<:-HEAP] may seem unsound at first blush. Typical type systems are flow-insensitive. In such systems, a reference has a *single* type over the entire scope in which it is defined, and hence, using covariant subsumption to unsafely "upcast" reference types can cause unsoundness. In our setting, covariant subtyping is sound as we treat the heap in a flow-sensitive manner. We assign different types to the current heap *before* evaluating an expression and the resulting heap *after* the expression has been evaluated. This allows a heap location to be updated to reflect a change in the type of the stored value, avoiding the aforementioned unsoundness.

Pure Typing Judgments The typing judgments for pure expressions are shown in Figure 7. The rules are quite standard [25, 14, 29, 4]. Note that the refinement predicates for these expressions precisely track the value of the expression. The only non-trivial rule is [T-PTR-ARITH] which handles pointer arithmetic. The refinement for the result uses the refinement $PAdd(\nu, a_1, a_2)$ (Section 2) which states that the value obtained by adding an offset a_2 to a base pointer a_1 yields an appropriately offset pointer into the same block. Recall that $BS(\nu)$ (resp. $BE(\nu)$) denotes the address where the block referred to by ν begins (resp. ends).

The rules [T-ARITH], [T-PTR-ARITH], and [T-PTR-COMP] use the index operators $+$ and $\sim$, which are binary operations on indices which approximate arithmetic operations and the comparison operator, respectively.

Typing Judgments The typing judgments for expressions and programs are shown in Figures 9 and 10. The program typing rules are straightforward. The expression typing judgment $\Gamma, h \vdash e : T/h'$ states that, in local environment Γ, if the heap initially satisfies h, then evaluating e produces a value of type T and a heap satisfying h'. The majority of the rules are straightforward; the most interesting rules are those that deal with memory access.

Both typing judgments are parametrized by a function γ mapping run-time heap locations r to corresponding locations ℓ in the static heap typing, used in [T-REF]. This function is a technical device used in the soundness proof, and is always $\emptyset$ when type-checking source code entered by the user, which cannot contain reference constants.

Type Well-Formedness $\boxed{\Gamma, h \vdash T}$

$$\frac{0 \leq n \qquad \Gamma; \nu : \langle n \rangle_i \vdash a}{\Gamma, h \vdash \{\nu : \langle n \rangle_i \mid a\}} \text{ [WF-INT]}$$

$$\frac{\ell \in \mathsf{Dom}(h) \qquad \Gamma; \nu : \mathtt{ref}(\ell, i) \vdash a}{\Gamma, h \vdash \{\nu : \mathtt{ref}(\ell, i) \mid a\}} \text{ [WF-REF]}$$

Abstract Block Well-Formedness $\boxed{\Gamma, h \vdash_A b}$

$$\frac{\begin{array}{cc}\Gamma, h \vdash T & Ind(n, T) \cap \mathsf{Dom}(b) = \emptyset \\ x \text{ fresh} & \Gamma; x : T, h \vdash_A b[x/@n]\end{array}}{\Gamma, h \vdash_A n : T, b} \text{ [WF-FIELD]}$$

$$\frac{\begin{array}{c}\Gamma, h \vdash T \\ Ind(n^{+m}, T) \cap \mathsf{Dom}(b) = \emptyset \qquad \Gamma, h \vdash_A b\end{array}}{\Gamma, h \vdash_A n^{+m} : T, b} \text{ [WF-ARRAY]}$$

Concrete Block Well-Formedness $\boxed{\Gamma, h \vdash_C b}$

$$\frac{\begin{array}{c}\Gamma, h \vdash T \\ Ind(i, T) \cap \mathsf{Dom}(b) = \emptyset \qquad \Gamma, h \vdash_C b\end{array}}{\Gamma, h \vdash_C i : T, b} \text{ [WF-CONCBLOCK]}$$

Heap Well-Formedness $\boxed{\Gamma \vdash h}$

$$\frac{}{\Gamma \vdash \mathtt{emp}} \text{ [WF-EMPTY]}$$

$$\frac{\begin{array}{cc}\tilde{\ell} \in \mathsf{Dom}(h) & \ell_k \notin \mathsf{Dom}(h) \\ \Gamma \vdash h & \Gamma, h * \ell_j \mapsto b \vdash_C b\end{array}}{\Gamma \vdash h * \ell_j \mapsto b} \text{ [WF-CONCRETE]}$$

$$\frac{\tilde{\ell} \notin \mathsf{Dom}(h) \qquad \Gamma \vdash h \qquad \Gamma, h * \tilde{\ell} \mapsto b \vdash_A b}{\Gamma \vdash h * \tilde{\ell} \mapsto b} \text{ [WF-ABSTRACT]}$$

World Well-Formedness $\boxed{\Gamma \vdash T/h}$

$$\frac{\Gamma, h \vdash T \qquad \Gamma \vdash h}{\Gamma \vdash T/h} \text{ [WF-WORLD]}$$

Schema Well-Formedness $\boxed{\vdash S}$

$$\frac{\begin{array}{c}x_1 : T_1 \ldots \vdash h \\ \text{for each } x_i, x_1 : T_1 \ldots x_{i-1} : T_{i-1}, h \vdash T_i \\ x_1 : T_1 \ldots \vdash T'/h'\end{array}}{\vdash \forall \ell \ldots (x_1 : T_1 \ldots)/h \rightarrow T'/h'} \text{ [WF-SCHEMA]}$$

Figure 6. Well-Formedness

4.2 Type Checking Memory Operations

Next, we discuss the rules for memory allocation, heap operations, function calls, and location unfolding. The key idea that enables our system to verify and infer invariants about in-memory data structures in the presence of temporary invariant violation is our distinc-

Pure Typing $\boxed{\Gamma \vdash_\gamma a : T}$

$$\frac{\Gamma \vdash_\gamma a : T_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2}{\Gamma \vdash_\gamma a : T_2}\ [\text{T-PureSub}]$$

$$\frac{0 \leq w}{\Gamma \vdash_\gamma \langle w \rangle_n : \{\nu : \langle w \rangle_n \mid \nu = \langle w \rangle_n\}}\ [\text{T-Int}]$$

$$\frac{\gamma(r) = \ell}{\Gamma \vdash_\gamma \mathtt{ref}(r, n) : \{\nu : \mathtt{ref}(\ell, n) \mid \nu = \mathtt{ref}(r, n)\}}\ [\text{T-Ref}]$$

$$\frac{\Gamma(x) = \{\nu : \tau \mid a\}}{\Gamma \vdash_\gamma x : \{\nu : \tau \mid \nu = x\}}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash_\gamma a_1 : \langle n \rangle_{i_1} \qquad \Gamma \vdash_\gamma a_2 : \langle n \rangle_{i_2}}{\Gamma \vdash_\gamma a_1 + a_2 : \{\nu : \langle n \rangle_{+(i_1, i_2)} \mid \nu = a_1 + a_2\}}\ [\text{T-Arith}]$$

$$\frac{\Gamma \vdash_\gamma a_1 : \mathtt{ref}(\ell, i_1) \qquad \Gamma \vdash_\gamma a_2 : \langle n \rangle_{i_2}}{\Gamma \vdash_\gamma a_1 +_p a_2 : \{\nu : \mathtt{ref}(\ell, +(i_1, i_2)) \mid \mathit{PAdd}(\nu, a_1, a_2)\}}\ [\text{T-Ptr-Arith}]$$

$$\frac{\Gamma \vdash_\gamma a_1 : \mathtt{ref}(\ell, i_1) \qquad \Gamma \vdash_\gamma a_2 : \mathtt{ref}(\ell, i_2)}{\Gamma \vdash_\gamma a_1 \sim a_2 : \{\nu : \langle W \rangle_{\sim(i_1, i_2)} \mid \nu = a_1 \sim a_2\}}\ [\text{T-Ptr-Comp}]$$

$$\frac{\Gamma \vdash_\gamma a : \{\nu : \mathtt{int} \mid \nu \neq 0\}}{\Gamma \vdash_\gamma \mathtt{assert}(a) : \mathtt{void}}\ [\text{T-Assert}]$$

Figure 7. Pure Typing Rules

tion between *concrete locations* and *abstract locations*. Thus, to better understand the rules for memory operations, we begin with a more thorough description of abstract and concrete locations.

Concrete Locations are names that refer to *exactly one* physical memory location. For example, a single item in a linked list has one physical location and thus can be identified with a concrete location. The block bound to a concrete location describes the current state of the contents of exactly one physical location.

Abstract Locations are names that refer to *zero or more* concrete locations. For example, all items in a linked list may share the same abstract location, although each item is at a different concrete location. The block bound to an abstract location is an invariant that applies to all elements which share that abstract location.

Since we wish to verify data structure invariants in spite of temporary invariant violation, we will allow memory to be accessed only through concrete locations. This will enable our type system to perform strong updates to the types of concrete locations, providing robustness with respect to temporary invariant violation. Because we wish to verify properties of unbounded collections, which are represented using abstract locations, we need a strategy to handle pointers to abstract locations.

Strategy for Collections We employ a two-pronged strategy for handling pointers to abstract locations, and thereby collections. First, as long as only a single pointer to an abstract location is used, we can be assured that only one corresponding concrete location is being accessed. We will use our *location unfold* operation to obtain a concrete location corresponding to a pointer's referent. As long as the abstract location is only accessed through this "unfolded" pointer, we can safely perform strong updates on the new concrete location. Second, if we must use another pointer to access the abstract location, we can no longer be assured that a single concrete location will be updated. When this happens, we will use the *location fold* operation to ensure that the contents of the concrete location created earlier meet the abstract location's invariant, disallow further use of the unfolded pointer (without another unfold), and allow the new pointer to be soundly unfolded.

Subtyping $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{i_1 \subseteq i_2 \qquad \mathsf{Valid}(\llbracket\Gamma\rrbracket \wedge \llbracket a_1 \rrbracket \Rightarrow \llbracket a_2 \rrbracket)}{\Gamma \vdash \{\nu : \langle n \rangle_{i_1} \mid a_1\} <: \{\nu : \langle n \rangle_{i_2} \mid a_2\}}\ [<:\text{-Int}]$$

$$\frac{i_1 \subseteq i_2 \qquad \mathsf{Valid}(\llbracket\Gamma\rrbracket \wedge \llbracket a_1 \rrbracket \Rightarrow \llbracket a_2 \rrbracket)}{\Gamma \vdash \{\nu : \mathtt{ref}(\ell, i_1) \mid a_1\} <: \{\nu : \mathtt{ref}(\ell, i_2) \mid a_2\}}\ [<:\text{-Ref}]$$

$$\frac{}{\Gamma \vdash \{\nu : \mathtt{ref}(\ell_j, i) \mid a\} <: \{\nu : \mathtt{ref}(\tilde{\ell}, i) \mid a\}}\ [<:\text{-Abstract}]$$

$$\frac{}{\Gamma \vdash \{\nu : \langle W \rangle_0 \mid a\} <: \{\nu : \mathtt{ref}(\tilde{\ell}, i) \mid a\}}\ [<:\text{-NullPtr}]$$

Block Subtyping $\boxed{\Gamma \vdash b_1 <: b_2}$

$$\frac{}{\Gamma \vdash \mathtt{emp} <: \mathtt{emp}}\ [<:\text{-Block-Empty}]$$

$$\frac{x \text{ fresh} \qquad \begin{array}{c}\Gamma \vdash T_1 <: T_2 \\ \Gamma; x : T_1 \vdash b_1[x/@n] <: b_2[x/@n]\end{array}}{\Gamma \vdash n : T_1,\ b_1 <: n : T_2,\ b_2}\ [<:\text{-Field}]$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash b_1 <: b_2}{\Gamma \vdash n^{+m} : T_1,\ b_1 <: n^{+m} : T_2,\ b_2}\ [<:\text{-Array}]$$

Heap Subtyping $\boxed{\Gamma \vdash h_1 <: h_2}$

$$\frac{\Gamma \vdash b_1 <: b_2 \qquad \Gamma \vdash h_1 <: h_2}{\Gamma \vdash h_1 * \ell \mapsto b_1 <: h_2 * \ell \mapsto b_2}\ [<:\text{-Heap}]$$

World Subtyping $\boxed{\Gamma \vdash T_1/h_1 <: T_2/h_2}$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash h_1 <: h_2}{\Gamma \vdash T_1/h_1 <: T_2/h_2}\ [<:\text{-World}]$$

Figure 8. Subtyping

In the following, we describe the typing rules for the key operations of location unfolding and folding and demonstrate how they allow us to soundly perform strong updates. We then describe the remaining heap-accessing operations: memory allocation, heap read and write, and function calls.

Unfolding The expression $\mathtt{letu}\ x = [\mathtt{unfold}\ \ell]\ a\ \mathtt{in}\ e$, which "acquires" a concrete pointer to the location $\tilde{\ell}$ that a points to, is typed by rule [T-Unfold]. The rule first typechecks a in Γ to determine where it points. The block b bound to this location is located in the initial heap, h, to find the invariant satisfied by the abstract location. With some modification, this same block is bound to a new concrete location, ℓ_j, to ensure that this concrete location initially satisfies the same invariants as the abstract location did.

The modification consists of a sequence of substitutions. The block b may contain types which reference previous elements by their indices (*i.e.,* may contain types containing names like $@i$). Such types only have meaning in the context of the block where the indices are bound; if the type is extracted from the block — by typing a read operation, for example — it will be meaningless, since the indices are not bound to types in the environment. To give these types meaning outside of the block, we create fresh variable names x_i for each non-sequence index i and extend the environment with appropriately-substituted bindings for these names. Each concrete location has a "selfified" refinement stating that the value at each index i is equal to the corresponding name x_i. Note that sequence indices are *not* bound to selfified types, because a sequence index binding represents multiple data values.

Finally, a pointer to ℓ_j is bound to x in the body e. Well-formedness checks ensure that no other concrete location corresponding to $\tilde{\ell}$ exists and that the new bindings do not escape the scope of the body.

Note that the pointer being unfolded must be non-null. Because null pointers are treated as references to arbitrary, possibly uninhabited, abstract locations with arbitrary invariants, allowing a null pointer to be unfolded would allow the introduction of arbitrary predicates into the environment, leading to unsoundness. By allowing only non-null pointers to be unfolded, we ensure that we only unfold pointers to concrete locations which had previously been allocated, initialized, and folded. Such pointers are guaranteed to genuinely satisfy the invariants of their abstract locations and so there is no risk of unsoundness in unfolding them.

Folding The expression $[\texttt{fold}\ \tilde{\ell}]$, which "releases" the concrete location currently assigned to $\tilde{\ell}$, is typed by rule [T-FOLD]. The rule uses subtyping to check that the concrete location ℓ_j satisfies the invariant specified by its corresponding abstract location $\tilde{\ell}$ and removes concrete location ℓ_j from the output heap, preventing further use of pointers to ℓ_j.

Memory Allocation The expression $\texttt{malloc}(\ell, x)$ is typed by rule [T-MALLOC], which creates a new concrete location corresponding to newly-allocated memory. The new concrete location corresponds to abstract location $\tilde{\ell}$, which is mapped to block b, giving the desired invariant for the new concrete location. This invariant is not yet established for the concrete location, which represents freshly-allocated memory; thus, the concrete location is mapped to $b^{\top}$, which is b with all refinements set to $true$, and it is up to the caller to establish the invariant. The expression returns a reference to the beginning of the concrete location (index 0); the refinement on this reference states that the reference is a safe pointer to the start of a block of size x, where safe is defined as

$$Safe(\nu) \doteq \nu \neq 0 \wedge BS(\nu) \leq \nu < BE(\nu)$$

The uniqueness of concrete location bindings within the heap is ensured using heap well-formedness; *i.e.,* if there is an active concrete location corresponding to the abstract location being allocated, it must be "folded up" before `malloc` is invoked.

Initial Heap The input heap of the program's main function gives the type of the heap at the start of execution. Since at the beginning of execution no locations have been allocated and no invariants established, this initial heap cannot contain concrete locations. It may, however, contain abstract locations, since they need not describe the contents of any concrete locations. Rule [T-MAIN], which typechecks the call to the main function, ensures the initial heap contains only abstract locations.

Pointer Read The expression $*x$ is typed by rule [T-READ]. This rule ensures that the pointer is valid; if so, the type of the read is given by the type bound in the heap at the reference's location, index pair. The heap is left unaltered.

Pointer Write The expression $*x_1 := x_2$ is typed by rules [T-WRITE-FIELD] and [T-WRITE-ARRAY]. If the reference identifies exactly one location within a block — *i.e.,* it has a singleton index n — the rule [T-WRITE-FIELD] can be used to return a new, strongly-updated heap where the type of the referent has been updated to the type of the value being assigned. Otherwise, a strong update is unsound; the rule [T-WRITE-ARRAY] is used to ensure that the new value has the same type as the previous value. Note that we could use fold/unfold to allow strong writes to arrays, but we eschew this for simplicity. Both rules ensure that the dereferenced pointer is valid.

Function Call The expression $[\ell \ldots]\ \texttt{f}(y \ldots)$ is typed by rule [T-CALL], which is inspired by the modular "footprint"-based frame rule from separation logic. This rule splits the initial heap into two portions: h_m, the portion of the heap which is modified by the function, and h_u, the portion of the heap which is left unmodified by the function. To ensure soundness, we check that h_m and h_u are individually well-formed; this prevents placing a concrete location in h_u and its corresponding abstract location in h_m, allowing the function to unsoundly unfold an already-unfolded location. The rule also generates a substitution mapping formal (location) parameters to actual (location) parameters. This substitution is used to check that the actual parameters and heap are subtypes of the formal parameters and heap. The result of the call is the return type and the function's output heap, both with the actual (location) parameters substituted for the formals. The resultant output heap is joined with the unmodified portion of the input heap to obtain the caller's heap after the function returns.

4.3 Type Soundness

We ensure the soundness of our type system by proving the standard progress and preservation theorems. Due to space restrictions, we can only present a high-level view of the soundness theorems; detailed proofs, as well as our standard call-by-value small-step semantics, can be found in [28].

We relate run-time heaps and heap types using a heap satisfaction relation $h \models_\gamma h^*$. Intuitively, this relation says that the run-time block assigned to r in run-time heap h has the block type assigned to $\gamma(r)$ in the heap type h^*.

Our transition relation takes an expression e, a run-time heap h, a heap type h_1, and a mapping from run-time heap locations to static heap type locations γ_1 and returns a new tuple of the same form representing the state of the program and typing of the heap after evaluating e. The transition relation is parametrized over a global environment Φ mapping functions to their definitions. We denote the single step transition relation by $\hookrightarrow_\Phi$ and use $\hookrightarrow_\Phi^*$ to denote its reflexive, transitive closure.

PROPOSITION 1. *(Preservation)*

$$\begin{array}{llll} \textit{If} & (e, h, \gamma_1, h_1) & \hookrightarrow_\Phi & (e', h', \gamma_2, h_2), \\ & \Phi, \emptyset, h_1 & \vdash_{\gamma_1} & e : T^*/h^*, \\ & h & \models_{\gamma_1} & h_1 \\ \textit{then} & \Phi, \emptyset, h_2 & \vdash_{\gamma_2} & e' : T^*/h^*, \\ & h' & \models_{\gamma_2} & h_2. \end{array}$$

PROPOSITION 2. *(Progress)*

$$\begin{array}{llll} \textit{If} & \Phi, \emptyset, h_1 & \vdash_{\gamma_1} & e : T^*/h^* \\ & h & \models_{\gamma_1} & h_1 \\ & & e \textit{ is not a value} & \\ \textit{then} & (e, h, \gamma_1, h_1) & \hookrightarrow_\Phi & (e', h', \gamma_2, h_2) \\ & \multicolumn{3}{l}{\textit{for some } e', h', \gamma_2, \textit{and } h_2.} \end{array}$$

(We elide well-formedness statements in the above for clarity and brevity; the details appear in the technical report [28].)

Type soundness implies the following safety properties: (1) all memory accesses occur on non-null pointers that are within the bounds of their allocated memory regions, and (2) no assertion failures occur at runtime.

4.4 Type Inference

Next, we give a brief overview of type inference in NanoC. Type inference occurs in three phases: the first infers Basic types for the program; the second inserts location fold and unfold operations where necessary; and the third infers refinement types using liquid type inference.

Basic Type Inference In previous work [29, 20], we based our type inference techniques on the rich type information provided by ML's type system. Because C programs are essentially untyped,

Expression Typing $\boxed{\Phi, \Gamma, h \vdash_\gamma e : T/h'}$

$$\frac{\Gamma \vdash_\gamma a : T}{\Phi, \Gamma, h \vdash_\gamma a : T/h} \text{ [T-PURE]}$$

$$\frac{\begin{array}{c}\Phi, \Gamma, h \vdash_\gamma e : T_1/h_1 \\ \Gamma \vdash T_1/h_1 <: T_2/h_2 \qquad \Gamma \vdash T_2/h_2\end{array}}{\Phi, \Gamma, h \vdash_\gamma e : T_2/h_2} \text{ [T-SUB]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\gamma a : \langle n \rangle_i \qquad \Phi, \Gamma; a \neq 0, h \vdash_\gamma e_1 : \hat{T}/\hat{h}' \\ \Phi, \Gamma; a = 0, h \vdash_\gamma e_2 : \hat{T}/\hat{h}'\end{array}}{\Phi, \Gamma, h \vdash_\gamma \texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2 : \hat{T}/\hat{h}'} \text{ [T-IF]}$$

$$\frac{\begin{array}{c}\Phi, \Gamma, h \vdash_\gamma e_1 : T_1/h_1 \\ \Phi, \Gamma; x : T_1, h_1 \vdash_\gamma e_2 : \hat{T}_2/\hat{h}_2 \qquad \Gamma \vdash_\gamma \hat{T}_2/\hat{h}_2\end{array}}{\Phi, \Gamma, h \vdash_\gamma \texttt{let } x \ = \ e_1 \texttt{ in } e_2 : \hat{T}_2/\hat{h}_2} \text{ [T-LET]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\gamma a : \{\nu : \texttt{ref}(\ell_j, i) \mid \mathit{Safe}(\nu)\} \\ h = h_1 * \ell_j \mapsto \ldots, i : T, \ldots\end{array}}{\Phi, \Gamma, h \vdash_\gamma *a : T/h} \text{ [T-READ]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\gamma a_1 : \{\nu : \texttt{ref}(\ell_j, n) \mid \mathit{Safe}(\nu)\} \\ \Gamma \vdash_\gamma a_2 : \tau \\ h = h_1 * \ell_j \mapsto \ldots, n : \{\nu : \tau \mid a\}, \ldots \\ h' = h_1 * \ell_j \mapsto \ldots, n : \{\nu : \tau \mid \nu = a_2\}, \ldots\end{array}}{\Phi, \Gamma, h \vdash_\gamma *a_1 := a_2 : \texttt{void}/h'} \text{ [T-WRITE-FIELD]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\gamma a_1 : \{\nu : \texttt{ref}(\ell_j, n^{+m}) \mid \mathit{Safe}(\nu)\} \\ \Gamma \vdash_\gamma a_2 : \hat{T} \\ h = h_1 * \ell_j \mapsto \ldots, n^{+m} : \hat{T}, \ldots\end{array}}{\Phi, \Gamma, h \vdash_\gamma *a_1 := a_2 : \texttt{void}/h} \text{ [T-WRITE-ARRAY]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\gamma a : \{\nu : \texttt{ref}(\tilde{\ell}, i_y) \mid \nu \neq 0\} \\ h = h_0 * \tilde{\ell} \mapsto i : T_i \ldots, i^+ : T^+ \ldots \\ \theta = [x_i/@i \ldots] \\ \Gamma_1 = \Gamma; x_i : \theta T_i \ldots \qquad x_i \text{ fresh} \qquad \ell_j \text{ fresh} \\ h_1 = h * \ell_j \mapsto i : \{\nu = x_i\} \ldots, i^+ : \theta T^+ \ldots \\ \Phi, \Gamma_1; x : \{\nu : \texttt{ref}(\ell_j, i_y) \mid \nu = a\}, h_1 \vdash_\gamma e : \hat{T}_2/\hat{h}_2 \\ \Gamma_1 \vdash h_1 \qquad \Gamma \vdash \hat{T}_2/\hat{h}_2\end{array}}{\Phi, \Gamma, h \vdash_\gamma \texttt{letu } x \ = \ [\texttt{unfold } \ell] \ a \texttt{ in } e : \hat{T}_2/\hat{h}_2} \text{ [T-UNFOLD]}$$

$$\frac{h = h_0 * \tilde{\ell} \mapsto \hat{b}_1 * \ell_j \mapsto b_2 \qquad \Gamma \vdash b_2 <: \hat{b}_1}{\Phi, \Gamma, h \vdash_\gamma [\texttt{fold } \ell] : \texttt{void}/h_0 * \tilde{\ell} \mapsto \hat{b}_1} \text{ [T-FOLD]}$$

$$\frac{\begin{array}{c}\ell_j \text{ fresh} \qquad h = h_0 * \tilde{\ell} \mapsto b \\ \Gamma \vdash h * \ell_j \mapsto b^\top \qquad \Gamma \vdash_\gamma a : \{\nu : \texttt{int} \mid \nu > 0\} \\ T = \{\nu : \texttt{ref}(\ell_j, 0) \mid \mathit{Safe}(\nu) \wedge \mathit{BLen}(\nu, n)\}\end{array}}{\Phi, \Gamma, h \vdash_\gamma \texttt{malloc}(\ell, a) : T/h * \ell_j \mapsto b^\top} \text{ [T-MALLOC]}$$

$$\frac{\begin{array}{c}\Gamma \vdash h_m \qquad \Gamma \vdash h_u \\ \Phi(\texttt{f}) = \cdot, \forall \ell_\texttt{f} \ldots . x_j : T_j \ldots / h_\texttt{f} \rightarrow T'/h'_\texttt{f} \\ \theta = [a_j \ldots / x_j \ldots][\ell \ldots / \ell_\texttt{f} \ldots] \qquad \Gamma \vdash h_u * \theta h'_\texttt{f} \\ \text{for each } j, \Gamma \vdash_\gamma a_j : \theta T_j \qquad \Gamma \vdash h_m <: \theta h_\texttt{f}\end{array}}{\Phi, \Gamma, h_u * h_m \vdash_\gamma [\ell \ldots] \ \texttt{f}(a_j \ldots) : \theta T'/h_u * \theta h'_\texttt{f}} \text{ [T-CALL]}$$

Figure 9. Expression Typing Rules

we first use a type inference pass to assign rich Basic types to local variables and expressions and to discover the types of the heap's contents.

Program Typing $\boxed{\Phi \vdash p : T/h}$

$$\frac{\begin{array}{c}\vdash \hat{S} \qquad \hat{S} = \forall \ell \ldots . x_j : \hat{T}_j \ldots / \hat{h} \rightarrow \hat{T}'/\hat{h}' \\ \Phi; \texttt{f} : \hat{S}, x_j : \hat{T}_j \ldots, \hat{h} \vdash_\emptyset e : \hat{T}'/\hat{h}' * h_0 \\ x_j : \hat{T}_j \ldots \vdash \hat{T}'/\hat{h}' \\ x_j : \hat{T}_j \ldots \vdash h_0 \qquad \Phi; \texttt{f} : \hat{S} \vdash p : T/h\end{array}}{\Phi \vdash \texttt{letf f} \ = \ \texttt{fun}(\texttt{x}_j \ldots)\{e\} : \hat{S} \texttt{ in } p : T/h} \text{ [T-FUN]}$$

$$\frac{h_0 \text{ abstract} \qquad \Phi(\texttt{f}) = ()/h_0 \rightarrow T/h}{\Phi \vdash \texttt{f}() : T/h} \text{ [T-MAIN]}$$

Figure 10. Program Typing

We first use the declared C types of all functions in the program to generate corresponding Basic type schemas. This process is largely automatic and rarely requires annotations to be added. The generated function schemas are then used to infer Basic types for local variables, expressions, and heap contents as follows: First, local variables and expressions are assigned types where the as-yet-unknown indices and locations are represented by variables. A system of subtyping and heap location inclusion constraints over these types is generated from the source program in a syntax-directed manner. Next, these constraints are simplified to a set of location equality (aliasing), index inclusion, and heap location inclusion constraints over the unknowns. Finally, the simplified constraints are solved using a fixed point algorithm to obtain solutions for the heap contents and the unknown index and location variables, giving the types of the local variables, expressions, and heap contents in the body of the function.

Location Fold and Unfold Inference Next, our system automatically inserts location fold and unfold expressions in order to ensure that every dereference is on a concrete pointer and that only one concrete location is unfolded at a time, as required by our typing rules. To do this, our system visits each block in the CFG of each function. Our system traverses the statements in the block in order, maintaining a list of which concrete location, if any, is unfolded for each abstract location. At the beginning of the block, there are no unfolded concrete locations; the sole exception is the entry block of a function, which may take a pointer to an unfolded location. At each dereference, our system checks if the dereferenced pointer points to the currently-unfolded concrete location for its abstract location. If not, our system inserts a fold to fold up the old concrete location, if any, and inserts an unfold operation on the dereferenced pointer, creating a new active concrete location which is assigned to this pointer. At the end of the block, all locations are folded.

Liquid Type Inference Finally, we use liquid type inference to infer refinement types and thus automatically discover data structure invariants. This step is similar to previous work [29, 20]; we give a brief outline here. As before, we observe that our type checking rules encode an algorithm for type inference and so we perform type inference by attempting to produce a type derivation. At various points in the derivation, we encounter types (resp. heaps, schemas) which cannot be synthesized directly from the form of the expression and the current environment but must be inferred. We insist that these types (resp. heaps, schemas) be *liquid*, denoted $\hat{T}$ (resp. $\hat{h}$, $\hat{S}$), *i.e.,* their refinements must be *liquid refinements* consisting of a conjunction of logical qualifiers. Whenever we encounter a type which must be inferred, we create a new *template type*, which is the Basic type inferred earlier where a fresh variable is used to represent the as-yet-unknown liquid refinement. We generate *subtyping constraints* over the template types using the subtyping premises in our type rules; the subtyping rules are used to reduce these constraints to simple *implication constraints* between refinement expressions and unknown refinement variables. These constraints are solved via abstract interpretation to yield a liquid refinement for each refinement variable. Replacing each variable with its solution yields a refinement typing for the program.

5. Evaluation

We implemented our type system in CSOLVE, a prototype static verifier for C programs. CSOLVE takes as input a C source file and a set of logical qualifiers, which CSOLVE uses to perform liquid type inference. CSOLVE outputs the inferred liquid types of functions, local variables, and heap locations and reports any refinement type errors that occur.

We applied CSOLVE to several challenging benchmarks, drawn from [19], [21], [26], and the example of Section 2, which illustrate common low-level coding idioms. The results are shown in Figure 11. In each case, CSOLVE was able to precisely reason about complex invariants of in-heap data structures and memory access patterns to statically verify memory safety by proving the absence of null pointer dereferences and array bounds violations. (In the case of `ft`, we show only array bounds safety; see Section 5.1.) We explain several of the benchmarks below.

String Lists Using CSOLVE, we verified the safety of a program implementing a C idiom for linked list manipulation which is particularly common in operating system code [9] and which requires precise reasoning about pointer arithmetic. Recall the example of Section 2, which contained functions for creating and initializing strings and for creating lists of strings. We add to that example the function `string_succ`, shown below, which takes a pointer to the `str` field of a `stringlist` and returns the next `string` in the list. (Explicit null checks checks have been omitted for brevity) This function is used in `init_succ`, which creates a list of several strings and initializes the second one using `init_string`. CSOLVE precisely tracks pointer arithmetic to verify `init_succ`, by proving that that the input to `init_string` has the type from Section 2.

```
slist *string_succ(string **s) {
1:slist *parent = (slist **)s - 1;
2:return parent->next->s;
}

void init_succ() {
  slist  *sl;
  string *succ;
  sl   = new_strings(3);
  succ = string_succ(&sl1->s);
  init_string(succ, '\0');
}
```

The `string_succ` function expects an argument `s` of type $\mathtt{ref}(\tilde{\ell}^1, 4)$ in a heap of the form

$$\tilde{\ell}^1 \mapsto 0{:}\mathtt{ref}(\tilde{\ell}^1, 0), 4{:}\mathtt{ref}(\tilde{\ell}^2, 0)$$
$$\tilde{\ell}^2 \mapsto 0{:}\{\nu{:}\mathtt{int} \mid 0 \leq \nu\}, 4{:}\{\nu{:}\mathtt{ref}(\tilde{\ell}^3, 0) \mid BLen(\nu, @0)\}$$
$$\tilde{\ell}^3 \mapsto 0^{+1}{:}\mathtt{char}$$

From Section 2, we know that the return type of `new_strings` provides a pointer of this type, assigned to `sl`, in the appropriate heap. Thus, we begin in `string_succ` with the assignment to `parent` on line 1. Since `s` is cast to a `stringlist`$*$, which is 4 bytes long, and decremented, the type of the pointer assigned to `parent` is $\mathtt{ref}(\tilde{\ell}^1, 0)$. Continuing on line 2, the type of `parent->next` is the same, since the `next` pointer points to a structure of the same type. Finally, the type of `parent->next->s` is given by the type at offset 4 of $\tilde{\ell}^1$, since `s` is the second item in the `stringlist` structure. Thus, `string_succ` returns a pointer of type $\mathtt{ref}(\tilde{\ell}^2, 0)$ — a pointer to a `string`— in a heap of the form shown above. This

Program	Lines	Qualifiers	Assumes	Time (s)
stringlist	72	1	0	2
strcpy	77	3	0	4
adpcm	198	13	0	42
pmap	250	3	0	34
mst	309	1	0	16
power	620	7	2	111
ft	652	2	6	310
ks	742	9	7	721
Total	2,920	39	15	1,240

Figure 11. Results. Lines is the number of source lines without comments. **Qualifiers** is the number of logical qualifiers used. **Assumes** is the number of manual assumptions inserted. **Time (s)** is the time in seconds CSOLVE requires to verify safety.

pointer is passed to `init_string`; as the pointer and heap meet the required invariants, CSOLVE verifies safety. Thus, CSOLVE precisely reasons about pointers and in-heap data structures and automatically verifies this example using the qualifiers $\mathbb{Q}$ from Section 2.

Audio Compression Using CSOLVE, we verified the memory safety of routines for ADPCM audio encoding and decoding. The encoder, outlined below, takes as input an audio stream consisting of an array of 16-bit samples and outputs a compressed stream using 4 bits to represent each sample. The encoder relies on complex loop invariants to ensure memory safety.

```
void encoder (int nsamples, short *in0, char *out0){
  short *in          = in0;
  char  *out         = out0;
  int    bufferempty = 1;
  char   buffer;
  for (int len = nsamples; 0 < len; len--){
    Read *in++;
    if (!bufferempty) {
      //Write to buffer elided
      *out++ = buffer;
    } else {
      //Write to buffer elided
    }
    bufferempty = !bufferempty;
  }
  if (!bufferempty) *out++ = buffer;
}
```

The encoder takes three parameters: `nsamples`, the total number of samples in the input; `in0`, a pointer to the start of the input buffer, an array of 16-bit `short` values; and `out0`, a pointer to the output buffer, an array of 8-bit `char` values. The number of elements in the input buffer is twice the number of elements in the output buffer. The pointer `in`, initially set to `in0`, is used to read data from the input buffer; the pointer `out`, initially set to `out0`, is used to write data to the output buffer. The `for` loop iterates through each element of the input buffer. At each iteration, the loop reads 16 bits (a single `short` value) from the input buffer and advances `in`. Each iteration also computes a new 4-bit value for the output; however, since `out` is a `char` pointer, the encoder must write 8 bits at a time. Thus, the encoder buffers output into a local `char` value and only writes to `out` every other iteration. The flag `bufferempty` indicates whether to write to and advance `out`. The final `if` writes to the output in case there is a value in the buffer which has not been written, *i.e.,* if there are an odd number of samples in the input.

CSOLVE verifies the safety of dereferences of `in` and `out`, by inferring that `in` and `out` have the respective types

$$\{\nu = \mathtt{in0} + \mathtt{nsamples} - \mathtt{len}\}$$
$$\{2 * (\nu - \mathtt{out0}) = \mathtt{nsamples} - \mathtt{len} - (1 - \mathtt{bufferempty})\}$$

which encode the crucial loop-invariants that relate the values of the respective pointers with the number of iterations and the flag. By inferring similar invariants CSOLVE verifies the decoding routine.

Virtual Memory Using CSOLVE, we verified the array safety of `pmap`, a 317-line program implementing a virtual memory subsystem of the JOS OS kernel [19] that comprises functions for allocating and freeing virtual address spaces, allocating and freeing a physical page backing a virtual page, and mapping two virtual pages onto the same physical page.

To ensure the safety of array accesses in `pmap` we must precisely reason about the values contained in the *collection* of *environment* structures that represents virtual address spaces. Each environment includes a mapping from virtual pages to physical pages, `env_pgdir`, represented as an array of fixed length. Each index of `env_pgdir` is mapped to either the physical page allocated to the virtual page or -1 if no physical page has been allocated. Environments are joined together in doubly-linked fashion to form a list of virtual address spaces.

The physical address space is described by an array of size N, `pages`. Operations like allocating and freeing physical pages use entries from an `env_pgdir` field to index into `pages`. Thus, to prove array safety, we must verify that the items in *every* `env_pgdir` in *every* environment are valid indices into `pages`. Formally, we must verify that every pointer to an environment points to a heap location $\tilde{\ell}$ whose description is

$$\tilde{\ell} \mapsto 0{:}\mathtt{ref}(\tilde{\ell}, 0); 4{:}\mathtt{ref}(\tilde{\ell}, 0); 8^{+4}{:}\{\nu{:}\mathtt{int} \mid \nu < N\}$$

where the pointers at offsets 0 and 4 are pointers to the next and previous environments, respectively, and the integers at indices in 8^{+4} are the entries in `env_pgdir`. Note that we cannot prove that every entry in `env_pgdir` is non-negative, as -1 is used to indicate an unused virtual page. However, every item in `env_pgdir` is verified to be non-negative before use as an index into `pages`.

Using CSOLVE, we were able to verify that the above heap typing holds and thus determine that every array access in `pmap` is within bounds. This is challenging because the majority of array accesses are indirect, using an entry in an `env_pgdir` field to index into an array of physical page data. This requires precise reasoning about the values of all elements contained in an in-heap data structure. Further, array offsets are frequently checked for validity in a different function from the one in which they are used to access an array, requiring flow-sensitive reasoning about values across function boundaries. Nevertheless, CSOLVE is able to verify the safety of all array accesses in `pmap`.

5.1 Limitations

In the following we discuss some limitations of our current system.

Flow-Sensitive Invariants Our system allows flow-sensitive strong updates to the type of a *single* member of an in-memory data structure. However, the type of the *whole* data structure is flow-invariant: each individual element must reestablish the data structure's type before the next member of the structure is accessed. For example, suppose that a list contains cells each of which has a data field with the value 0, and suppose that an loop iterates over the list and sets each data field to 1. Our system can only verify that at all points in time, each cell has the value 0 or 1. In particular, our system cannot determine that before (resp. after) the iteration, the data fields have the value 0 (resp. 1). We plan to extend our system with this kind of flow-sensitivity in future work.

Structural Subtyping Some C programs take advantage of *structural subtyping*: a function expects a pointer to a data structure and is called with pointers to "subtypes" of that structure which may contain additional fields that are not accessed by the function. Because the callee may modify some fields of the structure, it is not sound to keep the refinements on the untouched fields as they were before the call, since they may depend on the modified fields. On the other hand, eliminating these refinements could lead to unnecessary losses in precision when fields are read but not written. How to manage the combination of mutability, dependent refinements, and structural subtyping is left to future work.

Inserting Folds and Unfolds The heuristic for inserting location fold and unfold operations outlined in Section 4.4 is sometimes too conservative, particularly in its requirement that locations be folded before the entry and exit of a block. Consider the following code:

```
if (x->next != NULL) { assert(x->next != NULL); }
```

Because `x`'s location will be folded between the condition and the `assert`, the fact that `x->next` is non-null will be lost. This limitation prevents us from verifying the absence of null pointer dereferences in the `ft` benchmark. In future work, we aim to replace this heuristic with a more robust algorithm like that of [2].

6. Related Work

Static Dependent Types were first applied to formal verification in the context of mechanized proof assistants.In the late nineties there were projects that defined programming languages with restricted forms of dependent types. DML [32] showed how decidable checking could be achieved through the use of *indexed-types* and using a decidable logic for the indices. DML is a high-level language, and moreover, requires the user to provide manual annotations describing the types of recursive procedures and inductive datatypes. ATS [35] combines linear types with stateful views and explicit programmer-provided proof terms to specify and verify safety properties of an imperative language. In contrast to the above, we have previously demonstrated [29, 20] that for high-level languages the abstract interpretation enabled by Liquid Types can drastically reduce the annotations and automate verification. Our work brings those benefits to the low-level, imperative setting.

Dynamic Dependent Types offer an alternative to static verification where the hardest checks are deferred to run-time. Prior work [25, 14] explores dynamic and *hybrid* refinement types for higher-order functional languages. The DEPUTY system [10] implements hybrid dependent types for C. The DEPUTY type system was designed to track the information required to place appropriate run-time checks (`assert` statements) in the program. Thus, unlike LTLL, which is designed for static verification, the DEPUTY type system is flow- and path- insensitive, and oblivious to aliasing, heap updates and data structures. Further, unlike LTLL, the DEPUTY type system only supports a form of *local* type inference; users must write dependent type annotations for procedures. Once DEPUTY has placed the `assert` statements in the code, a precise static verifier like CSOLVE can discharge the checks at compile time.

Location-Sensitive Types encode pointer relationships within the type system and use the tracked information to determine the points where strong updates are possible. LTLL locations are inspired by the way in which locations are used to enable strong updates in [30], a system that was designed to type the machine code generated from a high-level language. Consequently, this system makes the assumption that *all* locations on the heap are concrete, which is not valid in the setting of low-level systems code. Our technique of using unfold and fold to allow temporary strong updates within an aliased collection is closely related to the notions of *restrict* [16, 2], *focus* [12], and *thawing and freezing* [1]. *Restrict* combines fold and unfold into a single lexically scoped operation, but this critically relies upon the existence of a high-level `new` operation that creates fully initialized locations. In contrast, LTLL requires a fold to add fresh locations returned by `malloc` to collections *after* they have been initialized. In this sense, the fold operation is a special case of the *adopt* [12] or *freeze* [1] operation that can be automatically inserted into low-level code *without* any programmer annotation. Finally, none of the above systems address the issue of pointer arithmetic; our approach of using blocks composed of fixed and periodic offsets is similar to that adopted by [31] in the context of dataflow-based alias analysis. Note that while tracking pointer arithmetic precisely is not essential to establish memory safety [11], it is essential to ensure the stronger invariants over fields that are inferred and verified by LTLL.

Floyd-Hoare Logic based verification techniques encode the entire machine state as monolithic logical predicates. These approaches are extremely expressive and precise, since arbitrarily complex specifications for collections can be encoded using universally quantified logical formulas. For the same reason, they can require significant manual intervention. Verification proceeds by composing the user-provided loop-invariants, pre- and post-conditions with the code to compute *verification conditions*. When possible, these conditions are discharged automatically [15, 9]. However, due to the brittleness of automatic quantified reasoning, one must sometimes resort to interactive theorem proving [23, 34, 13]. LTLL uses the underlying type system as a robust algorithm for quantifier generalization and instantiation, refinement predicates to achieve precision, and abstract interpretation to automate inference.

Abstract Interpretation based approaches to static verification fall into two categories. The first category includes extremely precise techniques for analyzing control-sensitive properties of individual variables [5, 3, 17, 7, 33, 18] which typically handle the heap very imprecisely. The second category includes extremely precise *shape analyses* that can characterize the heap using abstract domains tailored to the data-structures being analyzed [22, 27, 6, 8, 24]. In contrast, LTLL is an automatic technique that uses a combination of low-level types and predicate abstraction to compute invariants for data stored inside collections without using information about the shape of the underlying structure. In future work we would like to investigate ways to improve the precision of LTLL by enriching it with shape (or reachability) information, which would allow us to determine when a location has been *removed* from a collection.

7. Conclusion

In this paper, we broadened the scope of Liquid Types from the verification of pure functional programs in a high-level language to the verification of impure imperative programs in a low-level language. We did this by carefully combining several powerful constructs: precise models of typed heaps with concrete and abstract locations, a fold and unfold mechanism for coping with temporary invariant violation, and Liquid Types for inferring precise data structure invariants. We demonstrated that this combination enables the largely-automatic verification of memory safety properties through several realistic examples requiring precise reasoning about invariants of in-memory data structures.

Acknowledgments The authors wish to thank Domagoj Babic, Andrew Gordon, Ross Tate, and the anonymous reviewers for their detailed and insightful feedback.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, 2007.

[2] A. Aiken, J.S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI*, pages 129–140, 2003.

[3] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*. ACM, 2002.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[6] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[7] S. Chaki, J. Ouaknine, K. Yorav, and E.M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *SoftMC*, 2003.

[8] B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[9] J. Condit, B. Hackett, S. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, 2009.

[10] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[11] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. Ccured in the real world. In *PLDI*, pages 232–244, 2003.

[12] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*. ACM, 2002.

[13] J-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, 2007.

[14] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[15] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[16] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12. ACM, 2002.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*. ACM, 2004.

[18] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, pages 137–151, 2006.

[19] JOS. Jos: An operating system kernel. http://pdos.csail.mit.edu/6.828/2005/overview.html.

[20] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.

[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.

[22] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, LNCS 1824, pages 280–301. Springer, 2000.

[23] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, 2008.

[24] H. H. Nguyen, C. David, S. Qin, and W-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, 2007.

[25] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[26] The GNU Project. GNU coreutils. http://www.gnu.org/.

[27] Z. Rakamaric, J. D. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI*, 2007.

[28] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types: Technical report. http://pho.ucsd.edu/liquid.

[29] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[30] D. Walker and J.G. Morrisett. Alias types for recursive data structures. In *Types in Compilation 2000*, pages 177–206. Springer-Verlag, 2000.

[31] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, 1995.

[32] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

[33] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.

[34] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

[35] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *PADL*, pages 83–97. Springer, 2005.