Enforcing Alias Analysis for Weakly Typed Languages

Dinakar Dhurjati Sumant Kowshik Vikram Adve
University of Illinois at Urbana-Champaign
{dhurjati,kowshik,vadve}@cs.uiuc.edu

Abstract

Static analysis of programs in weakly typed languages such as C and C++ is generally not sound because of possible memory errors due to dangling pointer references, uninitialized pointers, and array bounds overflow. Optimizing compilers can produce unpredictable results when such errors occur, but this is quite undesirable for many tools that aim to analyze security and reliability properties with guarantees of soundness. We describe a compilation strategy for standard C programs that guarantees sound semantics for an aggressive interprocedural pointer analysis (or simpler ones), a call graph, and type information for a subset of memory. These provide the foundation for sophisticated static analyses to be applied to such programs with a guarantee of soundness. Our work builds on a previously published transformation called Automatic Pool Allocation to ensure that hard-to-detect memory errors (dangling pointer references and certain array bounds errors) cannot invalidate the call graph, points-to information or type information. The key insights behind our approach is that pool allocation can be used to create a run-time partitioning of memory that matches the compile-time memory partitioning in a points-to graph, and efficient checks can be used to isolate the run-time partitions. Furthermore, we show that the sound analysis information enables static checking techniques that *reliably* eliminate many run-time checks. We formalize our approach as a new type system with the necessary run-time checks in operational semantics and prove the correctness of our approach for a subset of C. Our approach requires no source code changes, allows memory to be managed explicitly, and does not use meta-data on pointers or individual tag bits for memory. Using several benchmarks and system codes, we show experimentally that the run-time overheads are low (less than 10% in nearly all cases and 30% in the worst case we have seen). We also show the effectiveness of reliable static analyses in eliminating run-time checks.

1. Introduction

Alias information, type information, and call graphs are the fundamental building blocks for many kinds of static analysis tools, including model checkers and error checking tools. For programs written in weakly typed languages, however, these fundamental building blocks may not be valid if the program performs any illegal memory operations such as array bound violations, dangling pointer dereferences, and references using uninitialized pointers, because these unsafe operations can overwrite memory locations in ways not predicted by the compiler. This means that even tools that aim to provide sound results with no false negatives [15, 8] cannot guarantee that they do so. In fact, software validation tools usually assume that such memory corruption cannot occur, e.g. memory allocations are assumed logically infinitely apart so that a buffer overflow cannot trample any other allocation. This problem is potentially important because many software validation tools today are used to detect security vulnerabilities or identify logical errors in important system software.

Unfortunately, it has proven extremely expensive to detect important classes of unsafe memory operations for a weakly typed language using static analysis, run-time checks or a combination of both [2, 30, 18, 25, 23]. All of these approaches have overheads that are prohibitively high for production use (e.g., 2x-11x). Furthermore most of these use heuristic techniques to detect certain errors, especially dangling pointer errors, and do not guarantee that all such errors will be detected.

An alternative approach is to use strongly typed systems that closely match the C type system, e.g., CCured [24] or Cyclone [13, 16]. The strong safety guarantees of these systems are technically attractive but they are obtained by disallowing explicit memory deallocation in general ([16] allows explicit deallocation in some restricted cases) and relying on automatic memory management. The adoption of automatic memory management for existing C software is likely to be slow for several reasons. First, it can take significant effort to tune legacy C programs to reuse memory effectively in a managed environment. Second, system and embedded software often have stringent requirements for performance, memory consumption, real-time constraints, and even power constraints. C has been widely used for such software partly because of the control it gives over performance and memory consumption. For these reasons (and because of the possible manual effort required to port programs to these languages), existing C and C++ applications may be slow to adopt such languages and many may not do it at all.

1.1 Overview of our approach

In this paper, we describe a novel, automatic approach an ordinary compiler can use to ensure three key analysis results — namely, a points-to graph, a call graph, and available type information — are sound, i.e., will not be invalidated by any possible memory errors, even undetected errors such as dangling pointer dereferences. Our solution builds on a previously published transformation we call *Automatic Pool Allocation* [22]. Automatic Pool Allocation uses the results of a pointer analysis to partition heap memory into fine-grain pools while retaining explicit deallocation of individual objects within pools, i.e., it partitions the heap but does not perform automatic memory management. The transformation was developed and used for optimizing memory hierarchy performance. The transformation essentially provides a run-time partitioning of the heap *that corresponds directly to the partitioning of memory in the points-to graph*.

The primary contribution of this work is to show how Automatic Pool Allocation can be used to enforce the validity of a given points-to graph despite potential memory errors, and to do so efficiently. This work is based on two key observations. First, by partitioning memory (at least) as finely as the points-to graph, we can check efficiently that a pointer does not reference a memory object that is not in its predicted points-to set. Second, many pools have a type-homogeneity property that allows us to eliminate many or most of these run-time checks. Furthermore, the type-homogeneity

property also allows us to statically ensure that dangling pointer errors will not cause any unexpected type violation.

There are four technical challenges that we solve in this work in order to use Automatic Pool Allocation for guaranteeing that the static analysis results are correct:

- We formalize the necessary properties of pool allocation as a new type system and the necessary run-time checks in an operational semantics so that we can prove the correctness of the overall. We give a formal proof of correctness for a subset of C that includes all important root causes of memory errors: dangling pointers, arbitrary casts and type mismatches, uninitialized variables, and array bounds violations.
- Second, pool allocation does not prevent dangling pointer references to freed memory. We show how to exploit type homogeneous pools to ensure that such dangling references do not cause any unexpected type violations in these pools. We have used this insight in previous work on enforcing memory safety, but only for a subset of C without pointer casts and we did not prove its soundness [10].
- Non-type-safe constructs in C (e.g., many pointer casts, unions, and varargs function calls) produce non-type-homogeneous pools. We show how to use run-time checks to enforce isolation of such pools from each other and from (statically checked) type homogeneous pools. We need additional run-time checks to detect other memory errors such as uninitialized references and array bounds violations for all pools.
- Finally, we show that we can use *sound* static analyses that exploit the points-to graph and call graph to *safely* optimize away many of the run-time checks and other run-time overheads. (We also give an example to show how a different static analysis tool (ESP [8]) could benefit from our approach.)

Our approach has several practical strengths and two key limitations. We discuss these briefly in Section 9.

We have implemented our techniques in a system we call SAFECode - Static Analysis For safe Execution of Code - using the LLVM compiler infrastructure [21]. Our system handles nearly the full generality of C, except programs with "manufactured addresses." We show experimentally using three groups of programs (Olden, Ptrdist, and three daemons) that the run-time overheads of SAFECode are close to 0 for most programs and less than 30% in all cases we have tested. We also show that the static analyses, whose correctness relies on alias analysis guaranteed by SAFECode, are important for achieving these low overheads.

The next section describes the language and analysis representations we assume as our inputs. Sections 3 describes our overall approach, type system, and operational semantics for a subset of C. The technical report gives a complete proof of soundness for this subset. Section 4 discusses the extensions to the type system to handle the full generality of C programs. Section 5 describes our implementation (SAFECode) of the type inference and run-time system. Section 6 describes static analyses that benefit from SAFECode. Section 7 presents our experimental evaluation of SAFECode. Section 8 discusses related work and section 9 concludes with a discussion of the strengths and limitations of the work.

2. Assumptions and Background

The inputs to our approach are:

1. a program written in C;
2. The results of a flow-insensitive, field-sensitive, unification-based pointer analysis on that program. As explained below, this includes both points-to information and type information for some subset of memory objects. The analysis may use various forms of context-sensitivity (see below).
3. A call graph computed for the program.

Our goal is to enforce the correctness of these analyses for all executions of the program. We do not concern ourselves with how these analysis results are actually computed; we only assume that these are given in the format described below. In our implementation, we use an analysis called Data Structure Analysis [20], a context-sensitive, field sensitive, unification based algorithm to compute both the pointer-analysis and the call graph. DSA is context-sensitive over entire acyclic call paths, both in its analysis and in the naming scheme for heap objects.

We include type information as part of the points-to representation because, in a weakly typed language like C, pointer analysis can still compute reliable type information for a subset of a program. DSA attempts to compute type information for every "points-to set" (defined more precisely below) in the program by inferring the intended type based on the *uses* of pointers to a points-to set object, and not based on the type declarations or cast operations in the program. Here, the "uses" of a pointer include indexing operations (&(x->Fld$_i$) and &E[E]), loads, stores, and indirect calls. If all pointers to a points-to set are used consistently as one type $\tau*$ (or as the appropriate type for a field within τ), then DSA infers the type of all objects in that set to be that type τ. Otherwise, DSA marks the type of the object to be "*Unknown*," explained below. Note that this approach of ignoring casts and only considering actual uses is able to infer types for heap objects allocated via the `malloc` operation in C, which is untyped: the (usual) cast of the returned pointer value is ignored, but any uses of the pointer are correctly considered. Experimental results for over 40 programs show that DSA can infer type information for the targets of 70% of load and store operations in most C programs (on average), and over 90% in many programs.

We will use a running example, shown in Figure 3, to illustrate the steps of our approach in the next few sections. Figures 3(a), 3(c) and 5 show this program respectively in original source form, after including the pointer analysis results as part of the program, and after conversion to our type system respectively. The second version (encoding the pointer analysis) is the input to our work in this paper. The syntax, typing rules and operational semantics for the final version are described in Sections 3.2, 3.3 and 3.4.

Although our implementation of SAFECode supports all of C, we use a subset of C as the source language in this paper to simplify the presentation. This language, shown in Figure 1, includes all sources of potential memory errors including pointers, structures, array operations, function pointers, arbitrary casts, dynamic memory allocation, and stack allocations. We only include 4-byte and 1-byte integer types (`int` and `char`) as primitive data types, and use distinct load and store operations for these types (load E for loading `int`s and loadc E for loading chars). The `cast` operation is similar to the one in C. We use a new operation called `alloca` (with arguments similar to `malloc`) to allocate memory on the stack. Also, global variables can only be pointers pointing to global memory that is allocated and initialized using a new operation called `galloc`, which takes a size parameter and an initializer; this essentially is how globals in a C program operate. These two features make it unnecessary to apply the & operator to get the address of a stack variable or global object; & is only used for indexing into structures, arrays and for function pointers.

2.1 Pointer Analysis Representation

Intuitively, the pointer analysis representation we assume can be thought of as a storage-shape graph [27, 17] (also referred to as a points-to graph) with the invariant that pointers pointing to two

vars		$x \;\; y$
Function names		f
Field names		Fld
Pointer Type pt	:=	$\tau*$
Function Type ft	:=	$\tau \longrightarrow \tau$
Structure Type st	:=	struct { $\text{Fld}_1 : \tau_i$, ..., $\text{Fld}_n : \tau_n$ }
Types τ	:=	int | char | pt | st | ft
declarations $decl$	:=	$\epsilon \mid \tau\ x; decl$
Statements S	:=	ϵ | S S | x = E; | store E, E; | storec E, E; | free(E); | if (E) then { S } else { S } | while (E) { S }
Functions F	:=	$\tau' f(x : \tau)$ { decl ; S }
Expressions E	:=	x | E op E | cast E to τ | load E | loadc E | malloc(E) | &E[E] | &(x->Fld_i) | &f | alloca(E) $op \in \{+, -, *, /, \%, \&\&, \|\|, \hat{}, <<, >>\}$
Definitions d	:=	F | struct x {Fld_i:τ_1, ..., Fld_n:τ_n }
GlobalDecl gd	:=	$\tau * \rho\ x$ =galloc(E, E)
Programs p	:=	$d\ \ p \mid gd\ \ p \mid \epsilon$

Figure 1. C like Language

Node var		ρ
new PointerType pt	:=	$\tau * \rho \mid \tau * (\rho, n)$
Function Sets fs	:=	f , fs $\mid \epsilon$
FuncPtrType fpt	:=	ft * fs
new StructType st	:=	st_{prev} |$\forall\rho$.st |τ<ρ>
new Type τ	:=	int | pt | st | ft | fpt | $Unknown$
new Expressions E	:=	E_{prev}|&(x->Fld_i) |&f
new Statements S	:=	S_{prev} | associate(ρ, τ)
new Definitions d	:=	d_{prev} | FSET fs = f, fs

Figure 2. Syntactic extensions for representing pointer analysis results. st_{prev}, E_{prev}, S_{prev} and d_{prev} are same as st, E, S and d in Figure 1.

different nodes in the graph are not aliased. Effectively, each node represents a set of memory objects created by the program and two distinct nodes represent disjoint sets of memory objects. We assume there is one points-to graph per function, since this allows either context-sensitive or insensitive analyses. Figure 3 (a) shows an example program in our language and Figure 3 (b) the associated storage-shape graph.

We use a type system to encode the results of points-to analysis (i.e., the storage shape graph) as type attributes within the program, using a type system analogous to Steensgaard's [27]. Each points-to graph node is encoded as a distinct type (although we continue to refer to nodes below). The input to our approach is a program in this type system, shown in Figure 2. Each pointer in this type system has a node attribute, ρ, describing the node it points to in the storage-shape graph. For example, in Figure 3(c), the type of `y` is `int*r2`, denoting that it points to objects of node `r2` in the points-to graph. Since a pointer can point into a structure at an offset $i > 0$, we use $\tau * (\rho, n)$ to denote the type of a pointer pointing to offset n (n is a compile type constant), within a struct of type τ at node ρ.

The statement `associate`(ρ, τ) associates node ρ of the graph with type τ, denoting that the node ρ contains objects of type τ. These objects may be pointers, say, `associate(r1,` τ' `*r2)`, and this directly encodes a "points-to" edge from node $r1$ to node $r2$. These attributes have redundant information. For example, we do not need the full type in an associate statement. `associate(r1, r2)` is sufficient to encode the edge between r1, r2; the full type can be easily gotten by traversing these edges. We chose this type system with redundancies as it simplifies the discussion in later Sections, and it is straightforward to take pointer analysis results from Steensgaard's like type system and convert it in to ours. Note that there can be only a single target node for each variable or field of pointer type, which is necessary for a unification-based analysis.

Memory that is used in a type inconsistent manner, e.g., via unions or casts in C, is assigned type $Unknown$, which is interpreted as an array of chars. In the running example, the target of z (node `r3`) has type $Unknown$ because this memory is accessed both as an `int` and as an `int**`.

The representation captures field-sensitive points-to information because each field (including pointer fields) is typed with distinct node attribute. This type system thus encodes a field sensitive pointer analysis. A struct type (with a pointer field) in a program can be used in different places with the field pointing to distinct sets of objects (e.g., when two linked lists are created with the same list node type). While it is possible to require a distinct struct type declaration for each different use of the struct type, we found that it is convenient to include polymorphic type constructors (similar to those used in Cyclone [13]), allowing a single struct type declaration that is instantiated it differently for different nodes in the graph based on the pointer target. As an example, $\forall\rho$. struct S { Fld : $\tau * \rho$ }, is a polymorphic struct type that can be instantiated to two different struct types with two different node attributes (e.g., struct S<ρ1>is the type for objects of type struct S in the original program and whose field points to node $\rho 1$ in the points-to graph while struct S<ρ2>is the type for those with field pointing to node $\rho 2$).

We represent the call graph in the input type system by adding a function set attribute called fs in Figure 2) to each function pointer type, making explicit the set of possible targets for that function pointer. The function set attribute can be initialized using the FSET definition. For example, the definition `FSET fs = func1, func2, func3` followed by a use `(int -> int)*fs fptr` denotes a function pointer `fptr` whose targets are the functions `func1, func2, func3`.

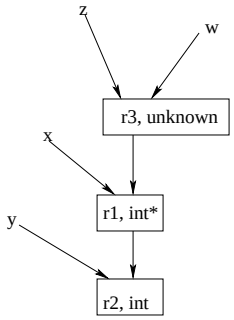
In the absence of `frees` and other memory errors, we can check that this program encodes the correct aliasing information by using typing rules similar to Steensgaard's, with necessary extensions to handle polymorphic struct types. We do not give those rules here as our approach described in Section 3 is stronger and subsumes this checking; we not only check that the static aliasing information is correct but we also enforce it in the presence of memory errors.

2.2 Background on Automatic Pool Allocation

Given a program containing explicit `malloc` and `free` operations and a points-to graph for the program equivalent to the description above, Automatic Pool Allocation transforms the program to segregate data into distinct pools on the heap [22]. By default, pool allocation creates a distinct pool for each points-to graph node representing heap objects in the program; this choice is necessary for the current work as explained later. Pools are represented in the code by pool descriptor variables. For a points-to graph node with $\tau \neq Unknown$, the pool created will only hold objects of type τ (or arrays thereof), i.e., the pools will be *type homogeneous* with a known type. We refer to these as TK (stands for type known) pools and all others as TU (stands for type unknown) pools. Calls to `malloc` and `free` are rewritten to call new functions `poolalloc` and `poolfree`, passing in the appropriate pool descriptor (details of the pool runtime library are omitted here for lack of space).

In order to minimize the lifetime of pool instances at runtime, pool allocation examines each function and identifies points-to graph nodes whose lifetime is contained within the function, i.e., the objects are not reachable via pointers after the function returns. This is a simple escape analysis on the points-to graph. The pool descriptor for such a node is created on function entry and destroyed on function exit so that a new pool instance is created every time the function is called. For other nodes, the pool

```
int **x, *y, *z, ***w, u;
x = (int **) malloc(4);
y = (int *)malloc(4);
z = (int *) malloc(4);
...
store y, x  // equivalent of *x = y
store 5 ,y
free(z) ; // creates a dangling pointer
store 10, z;
...
u = load z;  // equivalent of u = *z;
...
w = cast z to (int ***);
store x, w;
```




```
associate(r1, int * r2);
associate(r2, int) ;
associate(r3, Unknown) ;
int *r2 *r1 x;          int *r2 y;
int *r3 z; *r2*r1*r3 w;  int u;

x = malloc(4); y = malloc(4); z = malloc(4);

store y, x; store 5, y;
free(z);      // dangling pointer still exists
store 10, z ;
....
u = load z ;

w = cast z to (int *r2*r1*r3);
store x, w ;
```

Figure 3. (a) Original program, (b) its points-to graph, and (c) program with points-to graph encoded as types

descriptor must outlive the current function so pool allocation adds new arguments to the function to pass in the pool descriptor from the caller. Finally, pool allocation rewrites each function call to pass any pool descriptors needed by any of the potential callees.

In our previous work, we have used Automatic Pool Allocation to improve memory hierarchy performance [22] and to enforce memory safety without garbage collection in a type-safe subset of C [10]. The current work is the first to consider how automatic pool allocation can be used to enforce the correctness of a points-to graph, call graph and type information. Achieving this goal required solving several significant challenges, which were listed in the Introduction and will be discussed in the next few sections.

3. Type system

3.1 Overview

We first give an informal overview of our approach, focusing on four key insights we exploit in this work. The first two are new in the current work while the other two are borrowed from our previous work on memory safety for a type-safe subset of C [10].

The goal of our work is to ensure that memory errors (e.g., dangling pointer references after a free, array bounds violations, etc.) do not invalidate the points-to information, call graph, or type information computed by the compiler. The major challenge is enforcing points-to information; type information follows directly from this. The call graph is simply checked explicitly at each indirect call site (See Section 4.4 for a discussion on eliminating some of the run-time checks at indirect call sites).

Note that a node in a points-to graph (or the storage shape graph) is just a static representation of a set of dynamic memory objects. If these memory objects are scattered about in memory (as is usually the case), it is prohibitively expensive to check that a pointer actually points to a memory object corresponding to its target node (i.e., has not been corrupted by some memory error). As noted earlier, however, our transformation called *Automatic Pool Allocation* partitions the heap into regions based on a points-to graph [22]. That work focused on performance optimization and did *n*ot attempt to provide soundness guarantees on alias analysis or type information. That experience, however, led to the following new insight that is the key to the current work:

[**Insight1**]: *If memory objects corresponding to each node in the points-to graph are located in a (compact) region of the heap, we could check efficiently at run-time that the target of a pointer is a valid member of the compile-time points-to set for that pointer, i.e., that alias analysis is not invalidated.*

Note that this insight relies on the property that unaliasable memory objects are not allocated within the same region, which is not usually guaranteed by previous region-based systems [29, 13].

Non-heap (i.e., global and stack) objects may be in the same or different points-to sets as heap objects. We can simply include such objects in the set of address ranges for the appropriate pool (but many stack objects can be handled more efficiently as described in Section 4). Overall, the operation `poolcheck`($ph,\ A,\ o$) verifies that the address, A, is contained within the set of memory ranges assigned to pool, ph, and has the correct alignment for the pool's data type (or for the field at offset o if $o \neq 0$).

Even with the above partitioning of memory, checking every pointer dereference (or every pointer definition) would be prohibitively expensive. The second, relatively simple, insight allows us to eliminate a large number of the run-time checks:

[**Insight2**]: *Any initialized pointer obtained from a TK region or from an allocation site, will hold a valid address for its target region. All other pointers, i.e., pointers derived from indexing operations, and pointers obtained from TU regions (including function pointers), need run-time checks before being used.*

Intuitively, in the absence of dangling pointer errors and array indexing errors, an initialized pointer obtained from a TK region will always be valid; it cannot have been corrupted in an unpredictable way e.g. via arbitrary casts and subsequent stores (it would then be obtained from a TU region).

Uninitialized pointers, and array indexing errors are addressable via run-time checks. Dangling pointer references, however, are difficult to detect in general programs, and we do not attempt to detect or prevent such errors. Instead, we ensure that such errors do not invalidate the results of alias analysis, by exploiting two ideas that we also used in previous work on enforcing memory safety for a type-safe subset of C [9, 10]:

[**Insight3**]: *In a TK (type-homogeneous) region, if a memory block holding one or more objects were freed and then reallocated to another request in the same region with the same alignment, then dereferencing dangling pointers to the previous freed object cannot cause either a type violation or an aliasing violation.*

Essentially, we make sure that, if a dangling pointer to freed memory points into a newly allocated object, the old and new objects have the same static type and that any pointers they contain have identical aliasing properties. Thus loads or stores using the dangling pointers may give unexpected results but cannot trample memory outside the expected pool.

This principle allows free memory to be reused *within* the same region (unlike other region-based languages, which either disallow such reuse [29] or allow it only in restricted cases [16, 28]). For

Int		n, m, i, j, a
Region var		ρ
var		$x \quad y$
PointerType PT	::=	$\tau * \rho$
Types τ	::=	int | $Unknown$ | PT |handle(ρ, τ)
Statements S	::=	ϵ | S; S | x = E; | store E, E | storeU x, E, E | storec E, E | storecU E, E | poolfree(E, E) | poolinit(ρ, τ) x { S }; | pool{S}pop(ρ)
Expressions E	::=	var | V | E op E | load E | loadU x, E | loadc E | loadcU E | cast E to τ | castintpointer x, E to τ | poolalloc (x, E) | (x,&E[E])
Value V	::=	$Uninit$ | Int | region(ρ)
VarEnv VEnv	:	Var $\longrightarrow$ Value
Region R	::=	{ F ; RS }
RegionStore RS	:	Int $\longrightarrow$ Value
FreeList F	::=	ϕ | aF
LiveRegions L	:=	RegionVar $\longrightarrow$ RS
SystemHeap H		

Figure 4. Abstract Syntax for Core language

```
int *r2 *r1 x;
int *r2 y, u, tmp1;
int *r3 z; *r2*r1*r3 w;
poolinit(r2, int) r2handle {
  poolinit(r1, int *r2) r1handle {
    poolinit(r3, Unknown) r3handle {
      x = poolalloc(r1handle, 1);
      y = poolalloc(r2handle, 1);
      z = poolalloc(r3handle, 1);
      store y, x;        store 5, y;
      poolfree(r3handle,z);
      storeU r3handle, 10, z ;
      ...
      u = loadU r3handle, z ;
      ...
      w = cast z to (int *r2*r1*r3);
      storeU r3handle, x , w ; //type checks as region of r3
      ...                      //is Unknown
      tmp1 = loadU r3handle, w;
      v = castintpointer r2handle, tmp1 to int*r2 ;
} } }
```

Figure 5. Running example in our type system

reuse across regions, as we noted in our previous work [10], Automatic Pool Allocation already provides us a solution:

[Insight4]: *W*e can safely release the memory of a region when there are no reachable pointers into that region.

Overall, memory in a region can be reused within the region while the region is reachable and reused by any other region after the region is no longer reachable. In contrast, other region based systems only allow the latter reuse and not the former [29].

Finally, in order to prove the correctness of our approach, we formalize the key properties of our regions by extending the previous type system encoding points-to information (described in Section 2) in two ways: (1) to encode regions corresponding to points-to sets with allocation and deallocation out of these regions; and (2) to encode information about region lifetimes. The type system is designed to be mostly statically checkable for the correctness of encoded types (i.e. the points-to relations, lifetimes, and the call graph). We borrow a key idea from Tofte and Talpin's work on regions for ML [29] to simplify the type system, namely, we restrict region lifetimes to be lexically scoped (others have shown that this is not strictly necessary [1, 13]).

3.2 Syntax

To better illustrate our main idea, we limit our discussion here to a subset of the input language. This subset includes pointers, arbitrary casts, heap allocation and deallocation, array indexing and so has all the common sources of memory errors like dangling pointer dereferences, type cast errors, array bound overflow etc. Figure 4 gives the syntax for this subset along with new some constructs for encoding lifetimes of regions, region allocations, region handles, separating versions of `load/store` that require run-time check. The `associate` statement from the previous language is replaced by `poolinit` along with a lexical scope indicating where the association is valid, essentially creating a lifetime for the corresponding region. So regions in our system are nested, can only be created using the `poolinit` statement, and more importantly, unlike regions in Cyclone or TofteTalpin, a region can only contain objects of one type. As an example, the statement `poolinit`$(\rho, \tau)x_\rho\{$ S $\}$ creates a region named ρ that can hold objects of type τ, with the handle x_ρ. Our typing rules, described in Section 3.3, make it illegal to store an object of type other than τ in this region. The type of the region handle x_ρ is handle(ρ, τ). The lexical scoping, along with region attributes for pointers enable us to ensure that an object in a region can never escape the "scope" of the region. This is because the type for any pointer outside the scope of a region can not have the region name as an attribute; it won't type check as the the region name is not in scope. This allows us to type check the correctness of the inferred lifetimes of regions. Though this seems to disallow cycles in points-to graph, extensions for handling them are straightforward; we support creating multiple regions at the same lexical level (not present in the syntax here) and it can be used to create regions for all the nodes in a cycle at once.

The `malloc` from C is replaced by `poolalloc`. Intuitively `poolalloc` takes in a handle to the region as an argument and allocates an object (or an array of objects depending on the second argument) out of the region. The type of the allocated object(s) in a region is the type associated with the region. A *n*ovel feature in this language, not present in any other region based type systems is the `poolfree` statement, the equivalent of `free` from C. The `poolfree` statement frees a memory object and releases the memory to the region for further allocations out of the pool. $Uninit$ essentially represents the NULL value in C. We also have a new type of load instruction, `loadU` x, p that takes in a region handle x, a pointer p and whose semantics as described later require first checking if the pointer p points in to the region for which x is the handle and loads only if the check is successful. Similarly, we have `loadcU, storeU, storecU` versions of the `loadc, store, storec` respectively that perform run-time pool checks and only if successful peform the actual operations. `castintpointer` x, e `to` $\tau * \rho$, is a version of cast, that allows cast from `int` to pointer and requires a run-time pool check. The $\&p[x, e]$ is the pointer arithmetic from earlier, but now a requires a pool handle x as also as an argument as it needs to do a run-time check. Except for null pointer checks, all the new operations that take in a pool handle as an argument are the only operations that require a run-time check in our system. For all the other operations only checks performed are null-pointer checks.

Everything else in the syntax including pool{(S)}pop(ρ) , region(ρ) are not part of the source language but needed for operational semantics and are described in Section 3.4.

The Figure 5 shows the running example in this new syntax. The `associate` is now replaced by `poolinit` binding the lifetime of the pools. Some operations from the original example are replaced to use versions of the instructions that require a run-time

$$\text{(SS4)}\ \frac{C \vdash e : \tau * \rho \quad C \vdash \rho : \tau}{C \vdash \text{load } e : \tau}\ \tau \notin \{Unknown, \texttt{char}\}$$

$$\text{(SS5)}\ \frac{C \vdash e2 : \tau * \rho \quad C \vdash \rho : Unknown \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash \text{loadU } x, e2 : \texttt{int}}$$

$$\text{(SS6)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : int}{C \vdash \text{poolalloc}(x, e2) : \tau * \rho}$$

$$\text{(SS7)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e : int}{C \vdash \text{castintpointer } x, e \text{ to } \tau * \rho : \tau * \rho}$$

$$\text{(SS8)}\ \frac{C \vdash \tau' \quad C \vdash e : \tau * \rho}{C \vdash \text{cast } e \text{ to } \tau' * \rho : \tau' * \rho}$$

$$\text{(SS14)}\ \frac{C \vdash \rho : \tau \quad C \vdash e1 : \rho * \tau \quad C \vdash e2 : \tau}{C \vdash \text{store } e2, e1}\ \tau \notin \{Unknown, \texttt{char}\}$$

$$\text{(SS15)}\ \frac{C \vdash \rho : Unknown \quad C \vdash e1 : \tau * \rho \quad C \vdash e2 : \tau \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash \text{storeU } x, e2, e1}$$

$$\text{(SS16)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : \tau * \rho}{C \vdash \texttt{poolfree}(x, e2)}$$

$$\text{(SS17)}\ \frac{C \vdash \tau \quad \Gamma[x \mapsto \text{handle}(\rho, \tau)], \Delta[\rho \mapsto \tau] \vdash s}{C(= \Gamma, \Delta) \vdash \texttt{poolinit}(\rho, \tau) x\{s\}}\ x \notin \Gamma \text{ and } \rho \notin \Delta$$

Figure 6. Select Typing Judgments

check. The typing rules suggest which operations require run-time checks. Informally any load or store via a pointer to an $unknown$ region requires a run-time check and need to be replaced by the versions which perform the run-time check and so are casts from int to pointers, array accesses. Intuitively, we can think of inserting the appropriate run-time checks before those operations in the program.

3.3 Typing Rules

The type system is expressed by the following three judgments: $C \vdash e : \tau$ (for expression typing) , $C \vdash S$ (for statement typing), $C \vdash \tau$ (for type typing).

In these judgments C, the typing context, is a pair of typing environments $(\Gamma; \Delta)$ where Γ is a map between variable names and their types (built up using the variable declarations) and Δ is a map between region names and the type of objects stored in the region (built up using `poolinits`). We present in Figure 6 the typing rules that are unique to our approach. The full set of typing rules for the language in Figure 4 are included in Section 10.

(SS4) and (SS14) type loads/stores using pointers to type consistent memory (TK pools). They check that the type of the objects in the pool matches the type of the pointer operand. (SS5) is for loads using pointers to untyped $Unknown$ memory (TU pools); note that we get back an int. (SS7) allows a cast from int to pointer type, As discussed later in the operational semantics, such a cast requires a run-time check to make sure that the pointer is of the right type in the right pool. This coupled with (SS5) above enables loading pointers from TU pools safely. SS15) types stores to $Unknown$ memory. (SS8) types a cast from a pointer to region to another pointer pointing to the same region. This helps in supporting arbitrary casts of a pointer types as long as they have the same region attribute, *without* requiring any run-time check ((SS4) and (SS14) require that a pointer be cast back to the type of objects in the region before use). (SS17) is for creating a region using poolinit; we add the region variable and the handle to the typing environment before checking the body of the `poolinit` statement. (SS6) gives a type for the memory objects allocated in a pool. (SS16) frees objects only when they belong to the appropriate pool.

3.4 Operational Semantics

There are several sources of "programming errors" in weakly typed languages, including in the simple language described above, which can invalidate the results of any conservative alias analysis if we just rely on the the standard C semantics. For the language here, we list all the possible error sources below:

P1 dangling pointers to freed memory

P2 array bound violations

P3 accesses via uninitialized pointers

P4 arbitrary cast from an int type to another pointer type and subsequent use.

The one remaining source of memory errors in C, accessing a stack variable after a function returns, is not possible in our simplified language and is handled in Section 4.5.

In the case of **P1**, we have already explained how the type homogeneity principle allows us to reuse memory within a region while still making sure that dereferences to freed memory do not invalidate the aliasing properties. To implement the principle we need to maintain a list of freed memory objects within each region. Consequently our regions are defined as a tuple { F ; RS }, where F is a list of freed memory locations within the region, and RS is the region store (see Figure 4). Objects in the free list will be first used to satisfy allocation requests for the region. The region store, RS, is a partial map between memory addresses and their values.

VEnv, the variable environment, is a partial map between local variables and their values. We assume that region(ρ) is the handle for a region named ρ. The set of live regions, L, is a partial map between region names and the region store.

We present here only the operational semantic rules that are either novel to our approach or those that require a run-time check to solve the four problems above. See Figure 7 for the important rules and a brief explanation of each of them. Complete set of rules are available in Section 10.

The rules are described as a small-step operational semantics ($\longrightarrow_{rhs}$ for expressions and $\longrightarrow_{stmt}$ for statements). Each program state is represented by (VEnv, L, es) where VEnv is the variable environment (holding the values of local variables), L is the "stack" of regions that are live, and es is an expression or a statement in the program. A program state (VEnv, L, es) becomes (VEnv', L', es') if any of the semantic rules allow for it. The expression in the box denotes a run-time check that needs to be successful for the operation to be applicable.

Briefly, the four memory errors **P1**-**P4** are solved as follows. **P1** is solved using the type homogeneity principle, explained previously. This is implemented by rules **R14, R34** and the static typing rules that check operations on pointers to known-type pools. We detect problem **P2** using the run-time check on rule **R40**. To detect **P3**, we initialize all newly created memory and all local variables to $Uninit$ and check $Uninit$ pointers as shown in rules **R6, R14**. Issue **P4** is detected using **R31**.

More concretely, the rules shown in the figure work as follows:

R15,R17 Evaluating `poolinit` creates a new region, set the free list to be empty, and evaluates the body inside the syntactic construct pool{S}pop(ρ). This construct demarcates when the region is to be deallocated, viz., when the body (S) becomes empty. This is performed by rule R17.

R17 When all the statements within the pool construct are evaluated, we can destroy the pool (remove it from the stack of live regions.

R6 Performs a store via a type-consistent pointers, after checking that v_1 is not $Uninit$. $update(L, v1, v2)$ just updates the mem-

ory location $v1$ with value $v2$ (formally defined in Section 10). Loads via type consistent memory have a similar check for an uninitialized pointer.

R10 Performs a store via a pointer to $Unknown$ memory, after checking that the pointer value legally allows storing of a 4-byte value (an int).

R14 Frees an object from region, ρ, and adds it to the free list F of the same region.

R34 Returns a previously freed location from the free list. Note that this is where we rely on the type homogeneity principle to make error **P1** harmless.

R35 For a poolalloc, when the free list is empty, this requests fresh memory from the system. This is true if the system memory allocator is not buggy; i.e., the allocator falls within our trusted code base.

R31 A cast from int to another pointer type is always checked at run-time using a `poolcheck`, i.e., we check that the value is a (properly aligned) address in the appropriate pool for the pointer type, and if not, we abort. This solves problem **P4**. (Note that for structure types, the resulting pointer type specifies the field, i.e., $\tau * (\rho, n)$, needed for the alignment check.)

R40 For array indexing, we check that the resultant pointer after the arithmetic always points to the same pool as the source pointer. These run-time checks are not exact array bounds checks but a much coarser check for the pool bounds. This means some array bound violations may go undetected.

The complete list of run-time checks in our system are the checks in the boxes in Figure 7 along with checks on casts from integer to function pointers.

All these run-time checks do not require any metadata on individual pointer variables (usually required for precise array bounds checks) or runtime tag bits on any memory locations (usually required for RTTI or to track legal pointer values). Most importantly, we do not need to track the mapping of a pointer to its target pool descriptor because this mapping is fixed and known at compile time (because we use a unification-based pointer analysis, we have a 1-1 mapping of pool handles to points-to sets in each function). There are no run-time tag bits on any memory locations because valid pointer values are identified via poolchecks on `castintpointer` operations, and non-trivial type information is checked at compile time.

3.5 Soundness proof

Let $C \vdash_{env} (VEnv, L)$ denote the judgment for a well formed environment, the invariants of which are included in the technical report. Also assume that a run-time check failure leads to the Error state in the operational semantics. Then we can prove the following soundness theorem:

THEOREM 1. *If* $\Gamma \vdash S$ *and* $\Gamma \vdash_{env}$*(VEnv, L) then either (VEnv, L, S)* $\longrightarrow^*_{stmt}$*Error or (VEnv, L, S)* $\longrightarrow^*_{stmt}$ *(VEnv', L', ϵ) and* $C \vdash_{env}$ *(VEnv', L').*

Proof: The proof for this theorem is by induction on the structure of typing derivations (In Section 10).

The soundness result gives us with the following invariant – "For a well typed program containing pointer variable p whose declared type is $\tau * \rho$, in every execution state the value of p is guaranteed to be a pointer to an object in the region ρ". This holds even in the presence of undetected memory errors like dangling pointer dereferences and array bound violations, and thus it guarantees correctness of the aliasing information induced by our type system.

```
FunctionType ft   := τ → τ |τ → Unit|∀rho.τ.ft | ft<ρ>
new Types         := τ | ft |Unit
Functions F           :=   τ′f(x : τ) { S } |<ρ, τ> F
Instantiation finst   :=   f|x| finst <ρ>
new Statements := S
| poolinit(n, ρ1, ..., ρn, τ1, ..., τn)(x1, ..., xn) { S }
| call finst (x)
```

Figure 8. Syntactic extensions for remaining C constructs

3.6 Weaknesses

The key weakness of our system is that it permits dangling pointer errors and array bounds violations to go undetected (but confined within a pool). As explained in Section 1, the only solution to the former (for obtaining soundness guarantees) is via the use of automatic memory management. For the latter, although we could implement more precise array bounds checks, doing this efficiently would require some metadata on pointer variables. This introduces compatibility problems with external libraries, often can requiring manual changes by programmers. Our goal in this work is to provide a sound compiler approach that is *transparent* to programmers.

A second issue is that in some cases, our system might require more memory than the original C program (since we cannot free memory to the system until a region goes out of scope). In our previous work, we have evaluated the increase in the context of programs with no type casts and found that the increase is minimal in practice [10]. We believe this issue is unlikely to be significant in practice because we allow reuse within regions (which we believe is quite common for data structures that shrink and grow).

Finally, if the pointer analysis can not infer an allocation site and consequently a region, for a pointer (e.g. if the address is "manufactured" or read off the disk), we simply insert an `abort` before every use of such a pointer variable. This could reject a legal C program (other systems like CCured share the same limitation).

4. Extensions for full C

Several constructs of C were omitted in the previous section to explain our core ideas. Our type system and semantics correctly handle all constructs in C and so does our implementation. In this Section, we informally discuss how we handle the remaining constructs including function calls, function pointers and support for region polymorphic functions. Some of the ideas for implementing region polymorphism in functions and structs are directly borrowed from Cyclone [13]. However it is worth noting that our universal types are quantified only over region type variables (and not arbitrary type variables). This is sufficient in our domain of trying to retrofit polymorphic region types for otherwise non-polymorphic existing C code.

4.1 Structure types

The only extra safety implications of structure types are that (a) the `poolcheck` must use the offset o in checking alignment, and (b) structure indexing operations for pointers to TU regions need a poolcheck (similar to array indexing).A notational issue is that it is convenient to include polymorphic type constructors (similar to those used in Cyclone [13]) because a struct type with a pointer field can be used in different places with the field pointing to distinct sets of objects (e.g., when two distinct linked lists are created with the same list node type) The region-polymorphic structs for defining families of struct types. are exactly same as the "node" polymorphic structs described in Section 2. For example `struct S<rho> Field0 : int, Field1 : int * rho` defines a universal struct type parameterized over region variable

R6 (VEnv, L, store/storec v_2, v_1) $\longrightarrow_{stmt}$ (VEnv, update(L, v_1, v_2), ϵ) $\boxed{(v_1)! = Uninit}$

R10 (VEnv, L, storeU region(ρ), v_2, v_1) $\longrightarrow_{stmt}$ (VEnv, update(L, v_1, v_2, 4) }], ϵ) $\boxed{(v_1, v_1 + 3] \in \text{Dom(L}[\rho]\text{.RS)}}$

R14 (VEnv, L $\cup$ { (ρ, $\{F; RS\}$) }, poolfree(region(ρ), v)) $\longrightarrow_{stmt}$ (VEnv, L $\cup$ { (ρ, $\{vF; RS\}$) }, ϵ) $\boxed{v! = Uninit}$

R15 (VEnv, L, poolinit(ρ,τ)x{ S }) $\longrightarrow_{stmt}$ (VEnv $\cup$ {(x,region(ρ))}, L $\cup$ {(ρ, {ϕ; ϕ})}, pool{S}pop(ρ)) if ($\rho \notin$ Dom(L))

R17 (VEnv $\cup$ {(x,region(ρ))} , L $\cup$ {(ρ,R)} , pool{ϵ}pop(ρ)) $\longrightarrow_{stmt}$ (VEnv, L, ϵ)

R31 (VEnv, L, castintpointer (region(ρ), v to τ) $\longrightarrow_{expr}$ (VEnv, L, v) $\boxed{v \in \text{Dom(L}[\rho]\text{.RS)}}$

R34 (VEnv, L $\cup$ { (ρ, {a F; RS}) }, poolalloc(region(ρ), 1)) $\longrightarrow_{expr}$ (VEnv, L $\cup$ { (ρ, $\{F; RS\}$) }, a)

R35 (VEnv, L $\cup$ { (ρ, $\{\phi; RS\}$) }, poolalloc(region(ρ), 1)) $\longrightarrow_{expr}$ (VEnv, L[$\rho \mapsto \{\phi; RS[a \mapsto Uninit]\}$], a)
where a is a new address obtained from system allocator.

R40 (VEnv, L, (region(ρ), &$v1[v2]$)) $\longrightarrow_{expr}$ (VEnv, L, $v1 + v2 * \texttt{sizeof}(\tau)$) $\boxed{(v1 + v2 * \texttt{sizeof}(\tau)) \in \text{Dom(L}[\rho]\text{.RS)}}$
where τ is the "static" type of the individual element of the array, available from the declaration.

Figure 7. Select Operational Semantic Rules. Run-time checks are shown in boxes.

```
  <rho, int> int function func (handle<rho, int> ph,
                                      int*rho p)  {
    int q,z;
    int*rho r;
    r = poolalloc(ph, 1);
    q = load p;
    ...
    return z;
}
main() {
    poolinit(rho1, int) ph1 {
      poolinit(rho2, int*rho1) ph 2 {
       int *rho' p = poolalloc(ph1, 1) ;
       l1: call func<rho1>(ph1,p); //type correct
       l2: call func<rho2>(ph1,p); //type incorrect
       ...
    }
  }
}
```

Figure 9. Region Polymorphic functions

rho and whose `Field1` points to region `rho`. This universal type can instantiated with any region variable to get a new type.

4.2 Region-polymorphism for functions

We also support region polymorphic functions, parametrized via region names. An example is shown in Figure 9. Here the function `func` is declared as a region polymorphic function, which can be instantiated using a region variable whose type matches the expected type. The instantiation at l2 is incorrect since the type of the objects that can be allocated in the region do not match. Region polymorphism is necessary if we don't want to duplicate function definitions for each context in which they are used. Inferring this region polymorphism automatically for C programs based on points-to analysis has already been reported [22]. We leverage that work and only type check that the inferred polymorphism and its instantiation is correct.

4.3 Cycles in Points-to graphs

SafeRegionUse principle described earlier does not allow expressing mutually recursive types in our languages. This is clearly not acceptable for supporting general C programs. We solve this by requiring that the regions for mutually recursive data structures be created at the same lexical level. For this reason, we add a new construct to our language that enables creating multiple regions at the same lexical level. An example is shown in Figure 10. Here `poolinit` creates two regions `rho1, rho2`, such that `rho1` contains objects of type struct X¡rho2¿ and viceversa. Note that we only require that the region initialization (via `poolinit`) for these data structures to be done at the same lexical level, actual memory allocation with in a region is done at the same place as in the original program.

```
 struct Y<rho1, rho2>; //forward declaration
struct X<rho1, rho2> {
  Fld1 : struct Y<rho2, rho1>*rho2
}
struct Y<rho2, rho1> {
  Fld1 : struct X<rho1, rho2>*rho1 ;
}
...
poolinit(2, rho1, rho2,
    struct X<rho1, rho2>, struct Y<rho1, rho2>) (ph1, ph2)
{
 ....
}
```

Figure 10. Cycles in Points-to graph

4.4 Function pointers

In the discussion in the previous section, we have assumed that we could enforce the correctness of the input call graph by performing run-time checks at every indirect call site. A number of these run-time checks are unnecessary and can be eliminated using static analysis.

First, note that the input pointer analysis itself contains a conservative call graph obtained using a flow-insensitive unification based algorithm. However, the input call graph we must enforce could be more precise (at some call sites) than the call graph contained in the pointer analysis. Here, we first present simple typing rules that can statically check the call graph contained in the pointer analysis in most cases and add a few run-time checks where static checking is not possible. We then enforce the more precise input call graph by adding extra run-time checks only at call sites where more precision is required.

Recall that a function pointer has an extra type attribute called the function set attribute (see Figure 2) that identifies the set of functions that can be called via that function pointer [1] If two function pointers have the same fset attribute then they potentially

[1] This appears to limit the input call graph to be flow-insensitive for the input language described in Figure 1. Our implementation, however, first converts the program to single assignment form and with such a represen-

call the same set of functions. To differentiate between the input call graph and the call graph given by the pointer analysis, we use the attribute `PAFSET` to identify the function pointer targets given by pointer analysis.

Consider the code fragment below:

```
PAFSET fs1  = func1, func2, func3;
(int -> int)*fs1 f, g;
f = g; //type-checks since fset attributes are the same
```

Here `fs1` represents the set of functions { `func1, func2, func3` } . Our type checker checks that `func1, func2, func3` are compatible, that they have same number of arguments, compatible argument types and are alpha equivalent with respect to region type variables. For an array of function pointers, the `PAFSET` attribute should include all the functions that ever get stored to any location in the array. Assignments to function pointers require type equality. In the example program, `f = g` type checks because they have the same type attribute. Any casts from int type to a function pointer type, essentially function pointers that point to a TU pool or obtained from a TU pool, require a run-time check: we check at run-time if the function pointer actually points to one of the functions in the `PAFSET` attribute.

If the input call graph is more precise than the one given by the pointer analysis, then we need to add more run-time checks to account for the imprecision. As an example consider the case where the input call graph for the above code fragment, has a more precise `FSET` attribute for g, namely { `func1, func2` } . To enforce the more precise call graph we add run-time checks before any indirect call through g, to check that g points to one of { `func1, func2` } . Formally, if the `FSET` attribute of a function pointer in the input call graph and the `PAFSET` attribute of that function pointer (in the call graph obtained from points-to analysis) are different, then we insert a run-time check for the indirect calls through such function pointers.

4.5 Stack and Global Allocations

We have explained briefly in Section 2 how the the `alloca` expression in our input language can model stack objects and eliminate the & operator for taking address of stack objects in C. Pointers to stack locations are a possible source of memory errors; they can escape the function in which they are created, potentially allowing a dangling pointer dereference in a caller. To prevent this, we first pretend that all `alloca`s are actually `malloc`s and infer the life times of regions using automatic pool allocation just like before. If a stack object escapes a function, then the region for that stack object is created within all the callers and an argument is passed in. In this case, we have to allocate that stack object on the heap to avoid dangling pointers.

However, for those stack objects for which the regions are still created within the same function (i.e., not moved up to a parent function because they don't escape), if no run-time checks are needed for any pointer accesses pointing to these regions, our implementation allocates them on stack just like the original program since there are no frees in such regions. In practice we found that most stack allocations in original program do not escape, do not need run-time checks and can actually be allocated on the stack.

We support global allocations by pretending that they are heap allocations in the entry function, `main`, before any operation of the original program. Note that any pool corresponding to the global has to be a global pool and will be initialized in `main`. Since globals are never deallocated, our implementation, instead of allocating them from the heap area corresponding to the pool, allocates them in global data space just like the original program and only stores the range of addresses with the pool. A run-time pool check, which checks whether an address belongs to the pool, needs to compare with these global address ranges in addition to the heap allocations out of the pool.

4.6 Control flow

Finally, ordinary control does not require any additional safety checks. Adding typing rules and semantic rules for control flow to our language is fairly straightforward and we omit the discussion here.

5. Implementation

Our compiler system, SAFECode (*S*tatic *A*nalysis *F*or safe *E*xecution of *Code*), is implemented using the LLVM infrastructure [21]. In principle, SAFECode supports any source language translated into the LLVM IR, but most of our experience has been with C programs so far. SAFECode uses a context-sensitive pointer analysis algorithm called Data Structure Analysis (DSA) to compute the points-to graph, call graph, and type information [20].

Conceptually, analysis validation in SAFECode consists of a non-standard "type-inference" step using Automatic Pool Allocation, a standard type checking step, and insertion of the necessary run-time checks. These phases are described next, followed by a brief summary of the SAFECode run-time system.

5.1 Type Inference and Type Checking

The "type inference" phase of SAFECode takes the input program and the points-to graph computed by DSA and transforms the program to add the region type attributes and region parameters of our extended type system. This type inference is simply a direct application of Automatic Pool Allocation, described briefly in Section 2.2. The pools inserted by pool allocation correspond directly to the region types in our type system.

Because Automatic Pool Allocation is not a standard type inference algorithm, we use a separate (and standard) type checking phase to check its output. Because our type rules include the region types, region lifetimes, and lexical scoping of region parameters, our type checker effectively ensures the correctness of this region inference (including region lifetimes, region parameters, etc.). This takes Automatic Pool Allocation outside the trusted code base required in SAFECode. Effectively, our extended type rules are stronger than what is required to guarantee just the points-to information, call graph and source program type information.

Finally, the full list of run-time checks and operations required by the operational semantics include (i) "pool bounds" checks for all references using a pointer to or from an `Unknown`-type pool; (ii) pool bounds and alignment checks for array references to known-type pools (we also use an interprocedural static array bounds checking algorithm described in Section 6 to eliminate some of these checks); (iii) checks at an indirect call site if the function pointer value is obtained from an `Unknown`-type pool, checking that the target is a member of the callee set predicted by the call graph; and (iv) initialization of potential pointer fields within memory objects to `Uninit`. When a reserved address range is available (e.g., the high GB within a 4GB address space for Linux), we set `Uninit` to the base of this range so that the check is performed "for free" by the memory management hardware.

5.2 The SAFECode runtime system

The key new aspects of the run-time (and some relevant implementation details needed to understand them) are as follows.

A pool in our implementation is organized as a linked list of (large) blocks. The pool handle stores the header to this list. If

tation, the function set attributes are sufficient to encode a flow-sensitive version of the call graph.

there is insufficient space for a new allocation, the pool requests more blocks from the underlying system heap using malloc. An allocation request is satisfied by returning a free chunk within one block (or spanning multiple blocks if needed). One key change in the pool implementation is that heap metadata such as the object header describing the size of an allocated block and the free list cannot be interleaved with live objects in a pool since our approach allows some memory errors to overwrite arbitrary data within a pool. Allowing the metadata to be corrupted would potentially lead to arbitrary safety violations. We maintain metadata for the free list at the start of each free block and ensure (as part of the poolchecks below) that this data cannot be corrupted. To record the size of an allocated block so that it can be found efficiently, we take advantage of type homogeneity (which we have empirically found is available for most pools even in C programs, as explained in Section 2.1). We use a bit vector (with one bit per data element of the pool type) to track the start of each allocated object (or the start of a free chunk immediately after an allocated object). Because searching this bit vector would be very inefficient for large arrays, we allocate each large array in a (contiguous) set of new blocks and perform a `poolfree` for the array simply by freeing all the blocks. (The hash table used for poolchecks below allows us to identify quickly when a particular address is a large array.)

By far the most important operations (in terms of performance impact) are the pool bounds checks, which are used either during the array indexing or during cast operations. Given a memory address, this check verifies that the address is contained within the memory of the pool and has the correct alignment for the pool's data type. To make the check efficient, we request memory from the system in blocks of size of 2^k bytes for some fixed k. We maintain a hash set holding the starting addresses of all current blocks. Given an address to check, we compute the block holding the address by masking the low k bits and check if the block is in the hash set. If it is, then we check for the alignment criterion. A check, therefore, involves a mask, two loads, a hash lookup, and an alignment check.

We can eliminate many hash lookups by exploiting the high spatial locality exhibited by many memory references, especially array references. We use a one-element cache to remember the block address of the last successful hash lookup and alignment check. On each check, we first compare with this cached value; if successful, we can avoid the hash lookup. For example, for an array accessed sequentially, we only need a hash look up for one in every 2^k/(element-size) array accesses.

For uninitialized pointers, we are able to avoid a software check in many cases. Modern operating systems reserve a set of addresses for the kernel, e.g., Linux reserves the high GB of each process on a 32-bit machine. A user-level program accessing that address range would cause a hardware trap. We therefore set the $Uninit$ value to the base of reserved addresses, (and replace the constant '0' in any pointer-type expression with the same value), so that the hardware does the run-time check for us for free. This technique is unusable for kernel modules and also for references that may access a structure type with size greater than the reserved range (which is extremely rare). For these, we have to retain explicit software checks at run-time.

6. Sound Static Analyses Enabled By SAFECode

The semantic guarantees provided by our system can be used to write sound static analyses based on the points-to graph, call graph, and type information. In this section we first show that a static array bounds checking technique developed in our previous work, which relies on a call graph, can now be used soundly for non-type-safe programs in our environment. We also illustrate how our soundness guarantees about alias analysis can benefit other static analysis tools, using an existing software verification tool as an example.

```
void KernelEntryPoint(int **o) {
   int **q, *r;
   char arr[15];
1: r = malloc(....);
2: ... //some computation using r
3: free(r);
 if (o != NULL)
4:      q = o;
   else {
5:      q = malloc(..);
6:      *q = ... /* *q is initialized with some safe value */
   }
7: *r = ... /* dangling pointer error, this can overwrite *q */
8: if (o != NULL)
       Probe(o);   /* checks that *o is a valid pointer */
9: **q = data1; /* Dereference arbitrary pointer */
}
```

Figure 11. Example – Value flow analysis with memory errors

6.1 Static array bounds checking in SAFECode

We can use an interprocedural array bounds checking algorithm that we developed previously (for type-safe subset of C) to eliminate some runtime array bounds checks. The algorithm uses the call graph but not points-to-graph because it does not track values through loads/stores. It propagates affine constraints on integer variables from callers to callees (for incoming integer arguments and global scalars) and from callees to callers (for integer return values and global scalars). We then perform a symbolic bounds check for each index expression using integer programming (our compiler uses the Omega Library from Maryland [19]). We retain the run-time checks for all the array references that could not be proved to be safe using our static analysis. Since SAFECode semantics guarantee the correctness of the call graph, this optimization is safe (just like it would be safe for a type-safe language). To our knowledge, SAFECode is the first system for ordinary C programs (including explicit memory deallocation) where such an optimization can be performed safely.

Two other aspects of our system described earlier are actually also optimizations because they are not strictly necessary for correctness. These two are the use of bounded lifetimes for regions in Section 3 (which reduces memory consumption) and using stack allocations for non-escaping stack objects in Section 4.5 (which improves performance directly). These optimizations depend on alias analysis as well as the call graph, both of which are guaranteed sound.

6.2 Static Analyses in ESP

As final example, we briefly describe one software validation tool, ESP [8], that relies on alias analysis to give guarantees about programs and could benefit from the guarantees provided by our system. Although we explain this interms of ESP, other software validation tools could make use of our guarantees in a similar fashion.

ESP relies on *value flow analysis* [11], a static analysis used to identify the set of pointer expressions that refer to the memory locations that hold a certain value of interest, such as a lock. These sets are called *value alias sets* and computed by a data-flow analysis (*value flow simulation*) and transfer functions using May alias information provided by a flow-insensitive, unification-based context sensitive pointer analysis. This approach has been used to verify various properties in software, e.g., the Probe security property [11], which requires that any pointer passed into the kernel

from user space is checked ("probed") before being dereferenced by the kernel.

Consider applying ESP to verify the code fragment in Figure 11, which is a version of the kernel code fragment used in [11] modified to introduce a dangling pointer reference. In the function, `KernelEntryPoint`, the pointer `o` is passed in from a user routine and its target needs to be *probed* before being dereferenced by the kernel. Because of line 4, ESP tracks `q` and `o` as value aliases if `o != NULL`. The newly allocated memory when `o == NULL` is initialized to be *safe*. Lines 7 contains a memory error (a dangling pointer dereference). Since the system memory allocation could allocate previously freed memory of r for the allocation of `q`, this dangling pointer dereference could actually overwrite `*q`. This violates the results of the May-alias analysis that `q` and `r` are not aliased to each other. In line 8, the target of pointer `o` is probed. ESP thus transitions both the value aliases, `o` and `q`, to the *safe* state. Dereferencing `*q` is hence detected as safe by ESP. However, in reality, `*q` could now point to any location in memory and can be dereferenced by the program, violating the Probe security property. Enforcing the assumed aliasing properties is essential for the soundness of the tool.

In our system, the same example would allocate `q` and `r` in two different pools as they are not aliased. This makes sure that dangling pointer error in `r` is not allowed to trample the memory of `q`. While we do not detect the actual error, we make sure that the aliasing property is not invalidated.

The above is an example of a flow-sensitive program analysis that uses an external flow-insensitive alias analysis and can be easily made sound using our approach. For a more general flow-sensitive analysis that reasons about loads/stores, we must modify the semantics of malloc (and free) in the analysis so that the address returned by malloc may be "aliased" to any previously freed objects in the same alias set. This is a straightforward (and local) change within the implementation of a dataflow analysis.

7. Results

We present an experimental evaluation of SAFECode for several ordinary C programs and a few operating system daemons. These experiments have three goals:

- To measure the net overhead and different components of overhead incurred by our safety checking techniques;
- To evaluate the benefit of using sound static analyses enabled by SAFECode to eliminate various kinds of runtime checks.
- To compare the overhead of our approach to that of the CCured system.

7.1 Run-time Overheads

We evaluated our system using 10 programs from the Olden suite of benchmarks [6], 3 programs from PtrDist, and three system codes – bsd-fingerd-0.17, ftpd-BSD-0.3.2, and netkit-telnet-0.17 daemon. The benchmarks and their characteristics are listed in Table 1. We compiled each program to the LLVM compiler IR, perform our analyses and transformations, then compile LLVM back to C and compile the resulting code using GCC 3.4.2 at -O3 level of optimization. For the benchmarks we used a large problem size to obtain reliable measurements. For `ftpd` and `fingerd`, we ran the server and the client on the same machine to avoid network overhead, and measured their response times for client requests. We are successfully applied SAFECode to `netkit-telnetd` but this is an interactive program and we did not notice any perceptible difference in the response times. We do not report detailed timings for this code here.

The "native" and "LLVM (base)" columns in the table represents execution times when compiled directly with GCC -O3 and with the base LLVM compiler (without pool allocation or any SAFECode steps) again using the LLVM C back-end. Use LLVM (base) times as our baseline allows us to isolate the affect of the overheads added by SAFECode. The "native" column shows that the LLVM (base) code quality is comparable to GCC and reasonable enough to use as a baseline.

The "PA" column shows the time when we only run the pool allocator and do not insert any run-time checks, i.e., it shows the effect of pool allocation on execution time. The "PA + checks (except array)" column shows the execution time with all load/store checks are inserted except the checks for arrays. The "SAFECode" column shows the time with all run-time checks. The two ratios, SAFECode/LLVM and SAFECode/PA, show the net overhead of SAFECode relative to the base LLVM code without and with pool allocation.

The column "SAFECode/PA" shows that the run-time safety checks added by SAFECode have a relatively small impact on performance (over and above pool allocation): less than 10% in all cases except `ks` and `yacr2`, which have 11% and 18% overhead. The latter two overheads are entirely due to pool checks for array references, as seen by comparing the "PA+non-array checks" vs. the "SAFECode" columns.

Comparing the columns "SAFECode/LLVM" with "SAFECode/PA," we see that the pool allocation transformation has a significantly bigger impact on performance than the run-time checks. Four of the programs show significant slowdowns due to PA: `em3d`, `anagram`, `ks` and `yacr2`. We believe that these slowdowns are because our modified pool run-time library has not been tuned at all and, in fact, uses an inefficient bit-vector implementation of free lists. A more recent version of the pool runtime library used in [22] shows no slowdown for these four programs (and much more significant speedups for many other codes). We aim to merge our extensions with this version in the near future.

The `voronoi` benchmark fails at run-time due to the dereferencing of a pointer casted from an integer resulting in a pointer to an unknown memory object. We discovered that the program casts a pointer to an integer, performs complex arithmetic on the integer and then casts it back to a pointer. Our pointer analysis (DSA) tracks integers derived from pointers heuristically and did not capture this case, but can be extended to do so.

7.2 CCured Comparison

The last three columns in Table 1 compare the overhead of SAFECode with that of CCured, for the Olden benchmarks rewritten by the CCured team. We have not tried to compare our results on other system codes as it involved significant porting effort in writing the CCured wrappers. In all these programs SAFECode has significantly less overhead than CCured, even though SAFECode's pool checks are more expensive than the run-time checks inserted by CCured. The lower overhead can be attributed to the broad range of static analysis techniques employed by SAFECode for eliminating garbage collection (GC) overhead, stack safety checks, and many array bounds checks, and the run-time techniques that eliminate null pointer checks and metadata maintenance overhead. Note, however, that several of our static and run-time techniques for reducing overhead (except GC overhead) could be used with CCured as well. We believe that for end-users, any differences in the overheads of the systems is likely to be less important than the choice between automatic and explicit memory management.

7.3 Effectiveness of Static Analysis

Table 2 shows the effectiveness of our static checks and of segregating memory objects into TK and TU pools. Columns 2 and

Benchmark	Lines of code	execution times (secs) native	LLVM (base)	PA	PA + non array checks	SAFECode	CCured	Slowdown Ratios SAFECode /LLVM	SAFECode /PA	SAFECode /native	CCured /native
Olden											
bh	2053	1.449	1.357	1.338	1.361	1.403	1.923	**1.03**	1.05	0.97	**1.31**
bisort	707	11.740	11.530	11.531	11.531	11.531	11.358	**1.00**	1.02	0.98	**0.97**
em3d	557	13.960	11.29	14.245	14.245	14.248	20.812	**1.27**	1.00	1.02	**1.49**
health	725	1.909	1.936	1.296	1.296	1.299	1.710	**0.67**	1.00	0.68	**.90**
mst	617	11.259	12.920	12.837	12.837	12.96	16.956	**1.00**	1.01	1.15	**1.51**
perimeter	395	2.033	0.048	0.051	.051	0.051	2.544	**1.04**	1.00	.025	**1.25**
power	763	1.253	0.887	0.934	0.934	0.918	1.408	**1.03**	0.98	0.73	**1.12**
treeadd	385	5.426	5.457	5.425	5.425	5.425	14.784	**0.99**	1.00	1.00	**2.72**
tsp	561	1.277	1.270	1.250	1.250	1.250	1.578	**0.98**	1.00	0.98	**1.23**
voronoi	111	Rejected because of cast from integer to pointer									
System											
fingerd	338	6.410	6.555	6.617	6.617	6.753	-	**1.03**	1.02	1.05	-
ftpd	26653	1.210	1.185	1.160	1.160	1.190	-	**1.00**	1.03	0.98	-
PtrDist											
anagram	647	12.778	16.084	16.915	17.953	19.742		**1.23**	1.05	1.54	-
ks	782	3.554	4.429	4.501	4.501	4.981	-	**1.12**	1.11	1.40	-
yacr2	3982	3.795	3.991	4.398	4.398	5.204	-	**1.30**	1.18	1.37	-

Table 1. Benchmarks (telnetd in text) - Runtime Overheads

3 show the total number of static array accesses and the number that must be checked at run time. The next two columns show the total number of static loads and stores and the number that need to be checked at run time. The last two columns show the static number of **TU** and **TK** pools. We found that our static array safety checks were successful in eliminating some run-time array bounds checks in most programs. Our static pointer safety techniques eliminates the need for run-time load/store checks in all but 3 of the programs. In the remaining three, we have checks for loads/stores using a pointer to a **TU node**).

Benchmark	Static Counts Total array accesses	Checked array accesses	Total loads/ stores	Non-array pointer checks	TU	TK
bh	80	45	708	96	1	3
bisort	2	0	103	0	0	1
em3d	17	14	80	0	0	10
health	3	0	221	0	0	2
mst	4	3	53	0	0	5
perimeter	4	4	233	0	0	1
power	4	4	229	0	0	4
treeadd	2	0	31	0	0	1
tsp	0	0	176	0	0	1
fingerd	13	8	32	11	0	3
ftpd	362	209	1949	285	2	22
telnetd	432	363	1602	0	0	15
anagram	63	47	164	4	1	5
ks	58	52	326	0	0	3
yacr2	302	302	856	0	0	26

Table 2. Benchmarks - Effectiveness of Static Checks

8. Related Work

For weakly typed languages like C and C++, there are broadly two kinds of techniques addressing memory errors: memory safety techniques that try to detect or prevent some or all memory errors, and stronger approaches that provide soundness guarantees.

There have been a large number of systems for detecting memory access errors by adding run time checks and meta-data [14, 26, 2, 18, 25, 23, 30] (the work by Loginov at el. also detects type errors [23]). Except the Patil work [25], these systems use heuristic techniques that do not detect or eliminate all possible errors, especially dangling pointer errors which are quite difficult to detect reliably. Therefore, these systems do not provide a sound basis for static analysis techniques. The tool by Patil and Fisher [25] can reliably detect memory reference errors, including dangling pointer errors but at the cost of very high overheads (2x-6x in many programs). Furthermore, even this tool does not prevent type violations on references to legal memory addresses (though it might be extended to do so). Overall, none of these tools provide a sound semantics despite their high-overhead run-time checks.

Two systems, CCured [24] and Cyclone [13], both enable type-safe execution of C or modified C programs, which enables sound analysis of these programs. CCured ensures type-safe execution for standard C programs, with some source changes required for compatibility with external libraries. It uses a conservative garbage collector instead of explicit deallocation of heap memory. Compared with our approach, the major advantage of CCured is that it guarantees the absence of dangling pointer references. In contrast, a key contribution of our work has been to enable sound analysis while still retaining explicit memory management A second difference is that CCured introduces significant metadata for runtime checks. This metadata is the primary cause of the porting effort required for using CCured on C programs because it can require wrappers around some library functions. SAFECode uses no metadata on individual pointer values as explained in Section 3, and doesn't require complex wrappers.

There are also minor technical differences between the systems. Our classification of memory into type-consistent and $Unknown$ is analogous to the WILD and non-WILD types of CCured, except that we use a context-sensitive pointer analysis to infer the types of memory objects. We allow $Unknown$ memory to point to type consistent memory by performing a run-time check as explained in Section 3.1. CCured uses physical subtyping and RTTI to eliminate some run time overhead on pointer casts. Our type inference supports limited forms of physical subtyping (only for upcasts and casts from void* to other pointer types) but we plan to investigate a more sophisticated version in the future.

Cyclone [13, 16] uses a region-based type system to enforce strict type safety, and consequently enforces alias analysis, for a subset of C programs. Unlike SAFECode and CCured, Cyclone disallows non-type-safe memory accesses (e.g., operations that would

produce the equivalent of $Unknown$ type or WILD pointers). Cyclone and other region-based languages [4, 12, 5, 7, 28]) have two disadvantages relative to our work: (a) they can require significant programmer annotations to identify regions; and (b) either they provide no mechanism to free or reuse memory within a region (e.g., RTJava) or they allow deallocation of memory within a region only in special cases (e.g., uniqueness annotations to Cyclone [16] or reset region in ML kit for regions [28]). In all the above systems, data structures that must shrink and grow (with non-nested object lifetimes) should either be put into a separate garbage-collected heap or in regions when they use a restricted form of aliasing. In contrast, we infer the pool partitioning automatically with no annotations, and we permit explicit deallocation of individual data items within regions without aliasing restrictions or extra annotations. Reaps [3] provides regions without any soundness guarantees.

9. Concluding Discussion

This paper has described an approach to provide a semantic foundation (a points-to graph, call graph, and type information) for building sound static analyses for nearly arbitrary C programs. The approach can be added to any C compiler containing a pointer analysis that meets the specified properties (flow-insensitive, unification-based). The Automatic Pool Allocation transformation used in our work is quite straightforward — in fact, it is much simpler than a typical interprocedural pointer analysis — and (we believe) has significant value as an optimization in its own right [22]. The only additional effort required is to implement simple type-checking and check insertion, and some changes to the pool run-time system. Overall, we believe the strategy described here is relatively simple to add to a standard C compiler.

The approach also has some other practical strengths. First, it is fully automatic and requires no modifications to existing C programs. Second, it allocates and frees memory objects at exactly the same points as the original program, minimizing the need to tune memory consumption. Third, it supports nearly the full generality of the C language, except for programs that access "manufactured addresses" (which could also be supported via pragmas or compile-time options). Fourth, our experiments show that the run-time overheads of our approach are quite small, generally less than a few percent relative to code with pool allocation alone. We believe these overheads are low enough to be used in production code, especially when reliability or security is a significant concern.

The two major restrictions in this work so far are the requirements that the pointer analysis be flow-insensitive and unification-based. Our approach could be extended to a non-unification-based pointer analysis, mainly by extending Automatic Pool Allocation. The primary change would be to track at run-time which pool a pointer points to at any point in the execution. This requires some metadata for any pointer that may target multiple pools. With this extension, we believe that the semantic checking techniques (the type homogeneity principle, the pool runtime checks, and the optimizations of run-time checks) would apply directly.

Extending the techniques for a flow-sensitive alias analysis is more difficult but there are two reasons why this may not be a significant limitation in many situations. First, interprocedural pointer analysis algorithms used in practice are generally flow-insensitive because of the high cost of flow-sensitive whole program analysis [17]. Second, as discussed in Section 6.2, flow-sensitive techniques can be implemented (on top of flow-insensitive points-to results) in a sound manner using our approach.

References

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *PLDI*, June 1995.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, Orlando, FL, June 1994.

[3] E. Berger, B. Zorn, and K. McKinley. Reconsidering custom memory allocation. In *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 2002.

[4] G. Bollella and J. Gosling. The real-time specification for Java. *IEEE Computer*, 33(6):47–54, 2000.

[5] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *ACM Conf. on Prog. Lang. Design and Implementation*, 2003.

[6] M. C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.

[7] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *PLDI*, June 2004.

[8] M. Das, S. Lerner, and M. Siegle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, Berlin, Germany, Jun 2002.

[9] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Conf. on Language, Compiler, and Tool Support for Embedded Systems*, Jun 2003.

[10] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, Feb. 2005.

[11] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *Proc. of ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[12] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, pages 313–323, Montreal, Canada, 1998.

[13] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, June 2002.

[14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX*, 1992.

[15] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.

[16] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *Proc. of the 4th international symposium on Memory management (ISMM)*, 2004.

[17] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[18] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[19] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Computer Science Dept., U. Maryland, College Park, Apr. 1996.

[20] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.

[22] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI*, Chicago, IL, Jun 2005.

[23] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2001.

[24] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[25] H. Patil and C. N. Fischer. Efficient run-time monitoring using

shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.

[26] J. Seward. Valgrind, an open-source memory debugger for x86-gnu/linux.

[27] B. Steensgaard. Points-to analysis in almost linear time. In *ACM symposium on Principles of programming languages (POPL)*, 1996.

[28] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, 1997.

[29] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, pages 132(2):109–176, Feb. 1997.

[30] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

10. Full list of static and dynamic semantics rules

This section gives the complete list of static and dynamic semantic rules for the language shown in Figure 4.

10.1 Typing Rules

We first give the typing rules for the language. As noted in Section 3.3, the type system is expressed by the following judgments:

Expression typing: $C \vdash e : \tau$
Stmt typing: $C \vdash s$
Type typing: $C \vdash \tau$

In these judgments C represents the typing context and is composed of two type environments, denoted by C = Γ; Δ. Here Γ is the value type environment, essentially a map between variable name and the type of the variable and Δ is the region type variable environment, a map between region type variables and the type of the objects stored in the region.

Figures 11-13 show all the typing rules for our language. SS0-2 are similar to standard C typing rules. SS3 says that $Uninit$ can be any type. SS4 is similar to the standard C load via pointers not pointing to $Unknown$ region[2]. Note that SS4 ensures that the type of the pointer and the region to which a pointer points to are compatible. SS5 states that from a pointer to an $Unknown$ node, we can load only an `int`or a `char` but not a pointer. Since all casts from `int`to pointers require a run-time check, this ensures that pointers loaded out of TU memory are checked before usage at the time of the cast. SS6 gives a type for the memory objects allocated in a pool. SS7 allows a cast from an `int`to a pointer type. As discussed later in operational semantics, such a cast requires a run-time check to make sure that the pointer is of the right type in the right pool. SS8 types a cast from a pointer to region to another pointer pointing to the same region. This helps in supporting arbitrary casts of a pointer type to any other pointer type so long as they have the same region attribute. The use of a pointer in a load or a store or a call, however requires that such a pointer be cast back to the right type (see typing rules SS4, SS14). C programs often cast a pointer to a void * and then cast it back before using it. We avoid doing any run-time checks on these casts, while other approaches like CCured require extra run-time checks on these cast operations. SS9 is for array indexing. SS10 allows you to cast any type to an int.

SS11-13 are similar to standard C type rules. SS14 is for storing to a typed memory, SS15 is for storing to an $Unknown$ memory. SS16 is for type checking the free operation on an object from a pool– we can free an object from the pool only if the object belongs to the pool. SS17 is for creating a region using poolinit; we add the region variable and the handle to the typing environment before checking the body of the `poolinit` statement. SS18-21 are for judging whether a type is well formed.

Ignoring memory errors and the `free` operations, these typing rules actually check for the correctness of the input pointer analysis. This allows us to detect bugs or malicious pointer analysis inputs.

We now state a small lemma on our static type system that is used later in the soundness proof.

LEMMA 1. **(Well Formed Type lemma)**

If $C \vdash e : \tau$ *then* $C \vdash \tau$.

Proof: The proof of this lemma is straightforward induction on the structure of typing derivation. The only rules that introduce terms with new types into the system are (SS0, SS3, SS8, SS17) and all of them check that the new types are well formed *i.e.,* $C \vdash \tau$.

[2] For our extended language that includes structs, these typing rules may need to be modified slightly since we can load only primitive types (either an int or a char or a pointer) via load instruction. Loading an entire struct in a single load instruction is disallowed.

$$\text{(SS0)}\ \frac{C \vdash \tau \quad \Gamma(x) = \tau}{C(=\Gamma,\Delta) \vdash x : \tau} \qquad \text{(SS1)}\ \frac{}{C \vdash n : \text{int}}$$

$$\text{(SS2)}\ \frac{C \vdash e1 : \texttt{int} \quad C \vdash e2 : \texttt{int}}{C \vdash e1\ \ op\ \ e2 : int}$$

$$\text{(SS3)}\ \frac{C \vdash \tau}{C \vdash Uninit : \tau}\ \tau \neq \text{handle}(\rho', \tau')$$

$$\text{(SS4)}\ \frac{C \vdash e : \tau * \rho \quad C \vdash \rho : \tau}{C \vdash \text{load } e : \tau}\ \tau \notin \{Unknown, \texttt{char}\}$$

$$\text{(SS4char)}\ \frac{C \vdash e : \texttt{char} * \rho \quad C \vdash \rho : \texttt{char}}{C \vdash \text{loadc } e : \texttt{char}}$$

$$\text{(SS5)}\ \frac{C \vdash e2 : \tau * \rho \quad C \vdash \rho : Unknown \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash \text{loadU } x, e2 : \texttt{int}}$$

$$\text{(SS5char)}\ \frac{C \vdash e2 : \tau * \rho \quad C \vdash \rho : Unknown \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash \text{loadcU } x, e2 : \texttt{char}}$$

$$\text{(SS6)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : int}{C \vdash \text{poolalloc}(x, e2) : \tau * \rho}$$

$$\text{(SS7)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e : int}{C \vdash \text{castintpointer } x, e \text{ to } \tau * \rho : \tau * \rho}$$

$$\text{(SS8)}\ \frac{C \vdash \tau' \quad C \vdash e : \tau * \rho}{C \vdash \text{cast } e \text{ to } \tau' * \rho : \tau' * \rho}$$

$$\text{(SS9)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : \tau * \rho \quad C \vdash e3 : int}{C \vdash x, \&e2[e3] : \tau * \rho}$$

$$\text{(SS10)}\ \frac{C \vdash e : \tau}{C \vdash \text{cast } e \text{ to } \texttt{int} : int}\ \tau \neq \text{handle}(\rho, \tau')$$

Figure 12. Expression typing judgments

10.2 Assumptions

To keep our formalism tractable, we made several simplifications. We list some of them below.

- sizeof(int) = sizeof($\tau * \rho$) = 4 bytes
  We assume that the size of `int` and size of a pointer are same and is 4 bytes. This is not true on a 64-bit system, and the formalism and our run-time checks needed to be adjusted accordingly for such systems.
- The memory management algorithm used in the formalism is naive (does not do any compaction, on freeing an array object frees only the first element potentially leaking memory). We believe these issues are orthogonal to the main problem that we are trying to solve.

10.3 Operational Semantics

Figure 16 and Figure 15 contains all the operational semantic rules for the language in Figure 4. This expands on the select rules listed in Figure 7. The rules are split into two kinds of rules, one for expressions of the program (Figure 16) and another for statements of the program (Figure 15).

As noted in Section 3.4, the rules are described as a small-step operational semantics ($\longrightarrow_{rhs}$ for expressions and $\longrightarrow_{stmt}$ for statements). Each program state is represented by (VEnv, L, H, es) where VEnv is the variable environment (holding the values of local variables), L is the "stack" of regions that are live, H is the system heap (essentially the set of free addresses that are not used by any region in the program), and es is an expression or a

$$\text{(SS11)}\ \frac{}{C \vdash \epsilon}$$

$$\text{(SS12)}\ \frac{C \vdash s1 \quad C \vdash s2}{C \vdash s1; s2}$$

$$\text{(SS13)}\ \frac{C \vdash x : \tau \quad C \vdash e : \tau}{C \vdash x = e}$$

$$\text{(SS14)}\ \frac{C \vdash \rho : \tau \quad C \vdash e1 : \rho * \tau \quad C \vdash e2 : \tau}{C \vdash \text{store } e2, e1} \quad \tau \notin \{Unknown, \texttt{char}\}$$

$$\text{(SS14char)}\ \frac{C \vdash \rho : \texttt{char} \quad C \vdash e1 : \rho * \texttt{char} \quad C \vdash e2 : \texttt{char}}{C \vdash \text{storec } e2, e1}$$

$$\text{(SS15)}\ \frac{C \vdash \rho : Unknown \quad C \vdash e1 : \tau * \rho \quad C \vdash e2 : \tau \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash \text{storeU } x, e2, e1}$$

$$\text{(SS15char)}\ \frac{C \vdash \rho : Unknown \quad C \vdash e1 : \tau * \rho \quad C \vdash e2 : \texttt{char} \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash \text{storecU } x, e2, e1}$$

$$\text{(SS16)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : \tau * \rho}{C \vdash \texttt{poolfree}(x, e2)}$$

$$\text{(SS17)}\ \frac{C \vdash \tau \quad \Gamma[x \mapsto \text{handle}(\rho, \tau)], \Delta[\rho \mapsto \tau] \vdash s}{C(= \Gamma, \Delta) \vdash \texttt{poolinit}(\rho, \tau) x\{s\}} \quad x \notin \Gamma \text{ and } \rho \notin \Delta$$

Figure 13. Statement typing judgments

$$\text{(SS18)}\ \frac{}{C \vdash \texttt{int}} \qquad \text{(SS19)}\ \frac{}{C \vdash Unknown}$$

$$\text{(SS20)}\ \frac{\Delta(\rho) = \tau}{C \vdash \rho : \tau} \qquad \text{(SS21)}\ \frac{\vdash \rho : \tau}{C \vdash \tau * \rho}$$

$$\text{(SS22)}\ \frac{\Delta(\rho) = Unknown}{C(= \Gamma, \Delta) \vdash \tau * \rho} \qquad \text{(SS23)}\ \frac{C \vdash \rho : \tau}{C \vdash \text{handle}(\rho, \tau)}$$

$$\text{(SS24)}\ \frac{}{C \vdash \texttt{char}}$$

Figure 14. Well formed types

statement in the program. For notational convenience, we drop H from the program state as it is unchanged except for `poolinit` and `poolalloc`. A program state (VEnv, L, es) becomes (VEnv', L', es') if any of the semantic rules allow for it. The expression in the box denotes a run-time check that needs to be successful for the operation to be applicable.

A brief explanation of each of the rules is given below:

R1 For supporting sequential composition.

R2, R3 For assignment statements in the program.

R4, R5, R6 Performs a store via a type-consistent pointers, after checking that v_1 is not $Uninit$. update(L, v_1, v_2) just updates the memory location $v1$ with value $v2$.

R7, R8, R9 For progress on the operands of storeU/storecU instructions

R10 Stores a 4 byte value (integer or a pointer) via a pointer to $Unknown$ memory, after checking that the pointer value legally allows storing of a 4-byte value. The **Unknown** memory objects are organized as an untyped array of bytes. We will later show that v_1 always points to a valid address in the region because of run-time checks on casts and pointer arithmetic. However, since we are storing a 4-byte value, we need to make sure that a store is aborted, if v_1 points to last 3 bytes of some memory object and the bytes exceeding the object allocation are not a part of this region [3] .

R11 Stores a `char` via a pointer to $Unknown$ memory. Note that no-rune time check is needed since we later prove that v_1 points to a valid address in the region.

R12, 13, 14 Frees an object from region, ρ, and adds it to the free list F of the same region. The memory is retained in the pool for future allocations out of the pool and not released to the system. Note that the region store RS does not change, so any dangling pointer access continues to be valid.

R15 Evaluating `poolinit` creates a new region, set the free list to be empty, and evaluates the body inside the syntactic construct pool{S}pop(ρ). This construct demarcates when the region is to be deallocated, which is done when the body (S) becomes empty. Deallocating a region ρ releases all of RS back to the system heap.

R16 For progress within the syntactic pool construct.

R17 When all the statements within the syntactic pool construct are evaluated, we can destroy the pool (remove it from the stack of live regions).

R18 Evaluates a variable; gets the value of the variable previously stored in the VarEnv map.

R19, 20, 21 For C like arithmetic operations

R22, 23 Performs a load via type-consistent pointers, after checking that those pointers are not $Uninit$. getvalue(L, v) gets the value stored in address location v (This does not change the stack of regions in any way).

R24, 25 For progress on operands of loadU/loadcU instructions.

R26 Loads a 4-byte value via a pointer to $Unknown$ memory, after checking that the pointer value legally allows loading of a 4-byte value (an int) (similar to rule **R10**).

R27 Loads a `char` via a pointer to $Unknown$ memory. Note that this does not require a run-time check.

R28 Normal casts don't require any run-time check.

R29, R30 For progression operands of castintptr instruction.

R31 A cast from `int`to another pointer type is always checked at run-time. This solves problem **P4** in Section 3.4. In particular we check that the value is an address in the appropriate pool, if not, we abort.

R32, 33 For progress on the operands of poolalloc instruction.

R34 Returns a previously freed location from the free list. Note that this is where we rely on the type homogeneity principle to make a dangling pointer error harmless.

R35 For a poolalloc, when the free list is empty, this requests fresh memory from the system. The new address a must satisfy the invariant $\forall \rho \ \ a \notin$ Dom(L[ρ].RS), which says that the address a is not already used by any other region store.

R36 In case of array allocations, to keep the formalism tractable, we make a simplifying assumption that array allocations are always allocated using the system allocator. In our implementation, however, we compact our free lists where possible and allocate

[3] One way to avoid this run-time check is to pad each memory page of Unknown region by three extra bytes and ensure that the result of a pointer arithmetic or cast never points to the the three extra bytes.

out of the free lists if we find a contiguous free space that can hold the array. Similarly in rule **R14** we make a simplifying assumption that poolfree frees only one object (in case of arrays only one element of the array) and adds it to the free list. To support freeing of entire arrays, we just need to include the meta-data for storing allocation sizes in our formalism.

R37...40 Array accesses are checked at run-time; we check that the resultant pointer after pointer arithmetic always points to the same pool as the source pointer. Note that sizeof(τ) is a compile time constant, calculated from the declarations. *No run-time types (or RTTI) is used in our approach*

11. Soundness proof

We now give the proof of soundness for our type system. Section 11.1 discusses the invariants that we maintain in our system. Section 11.2 states the soundness theorem and gives its proof.

11.1 Invariants for well formed environments

In the rest of this discussion by environment we mean the pair (VEnv, L), the variable environment and the live region map in the heap. For an environment (VEnv, L), we define $||\tau||_{(VEnv,L)}$ to be as follows:

$||\texttt{int}||_{(VEnv,L)}$:= Int_{32}
$||\tau * \rho||_{(VEnv,L)}$:= $\{Uninit\} \cup$ Dom(L[ρ].RS)
$||\text{handle}(\rho, \tau)||_{(VEnv,L)}$:= { region(ρ)}
$||\text{Unknown}||_{(VEnv,L)}$:= Int_8
$||\texttt{char}||_{(VEnv,L)}$:= Int_8

From our earlier assumptions, sizeof(int) and sizeof($\tau * \rho$) is same: four bytes. This means that Dom(L[ρ].RS) $\subseteq \text{Int}_{32}$.

Intuitively for a well-formed type τ, $||\tau||_{(VEnv,L)}$ represents the set of values that a variable (or object) of that type can hold under that context and environment. We assume that $Uninit \in$ Int. We treat it as zero in our operations. For a pointer variable (or a memory location of pointer type), the values it can hold depend on the already allocated values in the region to which the pointer points to. For a pointer to region ρ only addresses in region ρ (or the uninitialized value) are legal values.

The judgement $\vdash_{env}$ stands for a well formed environment. An environment (VEnv, L) is well formed under a typing context C (denoted by $C(= \Gamma; \Delta) \vdash_{env}$ (VEnv, L)) if and only if the following invariants hold.

Inv1 Dom(Γ) = Dom(VEnv)

All variables in the typing environment are present in the variable environments and vice versa.

Inv2 Dom(Δ) = Dom(L)

All region names in the region type environment are already present in the domain of region maps and vice versa.

Inv3 $\forall x \in$ Dom(VEnv), if $C \vdash x : \tau$ then VEnv[x] $\in ||\tau||_{(VEnv,L)}$

If a variable has type τ, then it must contain only valid values of type τ. In particular, a pointer variable with region attribute ρ, must always point to an object in that region or it is not initialized.

Inv4 $\forall \rho \in$ Dom(L), if $C \vdash \rho : \tau$ then $\forall v \in$ Dom(L[ρ].RS), L[ρ].RS[v] $\in ||\tau||_{(VEnv,L)}$.

If region ρ is associated with type τ then each memory location in the region store will only contain values of the correct type.

Inv5 $\forall \rho \in$ Dom(L), L[ρ].F $\subseteq$ Dom(L[ρ].RS)

This invariant states that the memory addresses in the free list are a subset of the addresses of the region

Inv6 $\forall \rho_1 \rho_2 \in$ Dom(L), if $\rho_1 \neq \rho_2$ then Dom((L[ρ_1]).RS) $\cap$ Dom((L[ρ_2]).RS) = ϕ and $\forall \rho \in$ Dom(L), Dom(L[ρ].RS) $\cap$ H = ϕ.

A memory address cannot be part of two live regions. Also a memory address cannot be a part of system heap (i.e., unused by a program) and also a part of live region.

11.2 Proof

The proof of soundness is composed of two "invariant preservation" theorems — one for expressions and one for statements of the program. Since we have not included control flow in our formalization, all evaluations of expressions and statements terminate.

Notation: In the rest of this section, $\longrightarrow^*_{expr}$ represents the usual reflexive transitive closure of $\longrightarrow_{expr}$ and $\longrightarrow^*_{stmt}$ represents the usual reflexive transitive closure of $\longrightarrow_{stmt}$.

In order to prove the "invariant preservation" theorems, we make use of the lemmas listed in Figure 17, which are essentially big-step extensions of some of the small step rules given in Figures 15 and 16. The proof of each of the lemmas is by straightforward induction on the number of steps in the derivation of the hypothesis in that lemma. In the figure 17, we give the complete proof for the first lemma, proofs for the rest are similar and straightforward.

We now present three more lemmas, that are useful in proving the invariant preservation theorems.

LEMMA 2. **Update Lemma1**

If $C \vdash_{env}$ (VEnv, L), and $C \vdash \rho : \tau$ and if $v_1 \in$ Dom(L[ρ].RS), and $v_2 \in ||\tau||_{(VEnv,L)}$ then $C \vdash_{env}$ (VEnv, update(L, v_1, v_2)).

This lemma states that given a well formed environment, if we update a memory location in a region of the environment with the appropriate type then the resulting environment continues to be well formed.

Proof: From **Inv2** and $C \vdash \rho : \tau$, we have $\rho \in$ Dom(L).
So L = L' $\cup$ {(ρ,R)}
We also have $v_1 \in$ Dom(R.RS).
From the definition of update, we have update(L, v_1, v_2) = L' $\cup$ {(ρ, { R.F ;R.(RS[$v_1 \mapsto v_2$]) }) }
Now all the invariants except **Inv4** trivially hold. **Inv4** holds since $v_2 \in ||\tau||_{(VEnv,L)}$. q.e.d.

LEMMA 3. **Update Lemma2**

If $C \vdash_{env}$ (VEnv, L), and $C \vdash \rho : Unknown$ and if $[v_1, v_1 + 3] \in$ Dom(L[ρ].RS), and $v_2 \in ||Int||_{(VEnv,L)}$ then $C \vdash_{env}$ (VEnv, update(L, v_1, v_2, 4)).

Proof: The proof is straightforward extension of above; we just need to prove that byte function on Int_{32} gives an integer in Int_8, which is true from the arithmetic properties of integers.

LEMMA 4. **Getvalue Lemma1**
If $C \vdash_{env}$ (VEnv, L), $C \vdash \rho : \tau$ and if $v_1 \in$ Dom(L[ρ].RS), then getvalue(L, v_1) $\in ||\tau||_{(VEnv,L)}$.

Informally, this lemma states that given a well formed environment, if we load from a memory address in a region, the resulting value should have the type of the objects stored in that region.

Proof: We have ρ in Dom(L) from **Inv2**.
So L $\equiv$ L' $\cup$ {(ρ, R)}
From the definition of getvalue, we have getvalue(L, v_1) = L[ρ].RS[v_1]. Now from **Inv4** we have getvalue(L,v_1) $\in ||\tau||_{(VEnv,L)}$. q.e.d.

R1 $\dfrac{\text{(VEnv, L, S1)} \longrightarrow_{stmt} \text{(VEnv', L', S1')}}{\text{(VEnv, L, S1 ; S2)} \longrightarrow_{stmt} \text{(VEnv', L', S1' ; S2)}}$

R2 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, x = E)} \longrightarrow_{stmt} \text{(VEnv', L', x = E')}}$

R3 (VEnv, L, x = v_1) $\longrightarrow_{stmt}$ (VEnv[x $\mapsto$ v_1], L, ϵ)

R4 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, store/storec E, E}_2\text{)} \longrightarrow_{stmt} \text{(VEnv', L', store E', E}_2\text{)}}$

R5 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, store/storec } v\text{,E)} \longrightarrow_{stmt} \text{(VEnv', L', store } v\text{, E')}}$

R6 (VEnv, L, store/storec v_2, v_1) $\longrightarrow_{stmt}$ (VEnv, update(L, v_1, v_2), ϵ) $\boxed{(v_1)! = Uninit}$
where
update(L, v_1, v_2) := L' $\cup$ {(ρ, { R.F ;R.(RS[$v_1 \mapsto v_2$]) }) } if $\exists \rho \in$ Dom(L) s.t. L = L' $\cup$ {(ρ,R)} and $v_1 \in$ Dom(R.RS)
L else

R7 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, storeU/storecU E, E}_2\text{, E}_3\text{)} \longrightarrow_{stmt} \text{(VEnv', L', storeU/storecU E', E}_2\text{, E}_3\text{)}}$

R8 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, storeU/storecU } v_1\text{, E, E}_3\text{)} \longrightarrow_{stmt} \text{(VEnv', L', storeU/storecU } v_1\text{, E', E}_3\text{)}}$

R9 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, storeU/storecU } v_1\text{, } v_2\text{, E)} \longrightarrow_{stmt} \text{(VEnv', L', storeU/storecU } v_1\text{, } v_2\text{, E')}}$

R10 (VEnv, L, storeU region(ρ), v_2, v_1) $\longrightarrow_{stmt}$ (VEnv, update(L, v_1, v_2, 4) }], ϵ) $\boxed{(v_1, v_1 + 3] \in \text{Dom(L[}\rho\text{].RS)}}$
where
update(L, v_1, v_2, 4) := L' $\cup$ {(ρ, { R.F ;R.(RS[$v_1 \mapsto$byte(v_2, 3)][(v_1+1) $\mapsto$byte(v_2, 2)][(v_1 + 3) $\mapsto$byte(v_2, 1)][(v_1 + 4) $\mapsto$byte(v_2, 0)])
if $\exists \rho \in$ Dom(L) s.t. L = L' $\cup$ {(ρ,R)} and $[v_1, v_1 + 3] \in$ Dom(R.RS)
L else
and byte(n, k) := (n << (8 * (3 - k))) >> 24 .

R11 (VEnv, L, storecU region(ρ), v_2, v_1) $\longrightarrow_{stmt}$ (VEnv, update(L, v_1, v_2) }], ϵ)

R12 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolfree(E, E}_2\text{)} \longrightarrow_{stmt} \text{(VEnv', L', poolfree(E', E}_2\text{)}}$

R13 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolfree(}v\text{, E)} \longrightarrow_{stmt} \text{(VEnv', L', poolfree(v, E')}}$

R14 (VEnv, L $\cup$ { (ρ, {F; RS}) }, poolfree(region(ρ), v)) $\longrightarrow_{stmt}$ (VEnv, L $\cup$ { (ρ, {vF; RS}) }, ϵ) $\boxed{v! = Uninit}$

R15 (VEnv, L, poolinit(ρ,τ)x{ S }) $\longrightarrow_{stmt}$ (VEnv $\cup$ {(x,region(ρ))}, L $\cup$ {(ρ, {ϕ; ϕ})}, pool{S}pop(ρ)) if ($\rho \notin$ Dom(L)).

R16 $\dfrac{\text{(VEnv, L, S)} \longrightarrow_{stmt} \text{(VEnv', L', S')}}{\text{(VEnv, L, pool\{S\}pop}(\rho)\text{)} \longrightarrow_{stmt} \text{(VEnv', L', pool}\{S'\}\text{pop}(\rho)\text{)}}$

R17 (VEnv $\cup$ {(x,region(ρ))} , L $\cup$ {(ρ,R)} , pool{ϵ}pop(ρ)) $\longrightarrow_{stmt}$ (VEnv, L, ϵ)
Note that H the set of addresses in the system heap and not used by the program gets updated by H $\cup$ Dom(R.RS)

Figure 15. Operational Semantic Rules for Statements

LEMMA 5. **Getvalue Lemma2**
If $C \vdash_{env}$ (VEnv, L), $C \vdash \rho : Unknown$ and if $v_1 \in$ Dom(L[ρ].RS), then getvalue(L, v_1,4) $\in ||Int||_{(VEnv,L)}$.

Proof: Proof is straightforward extension of above; we just need to prove that the combine function on Int_8 gives an Int_{32}, which is true from the arithmetic properties of integers.

LEMMA 6. ***(*Safe region deallocate Lemma*)***

If (Γ; Δ) $\vdash \tau$, and $x \notin$Dom(Γ), and $\rho \notin$Dom(Δ), and ($\Gamma[x \mapsto$handle(ρ, τ')]; $\Delta[\rho \mapsto \tau']$) $\vdash_{env}$ (VEnv $\cup$ {(x,region(ρ))}, L $\cup$ {(ρ,R)}) then $||\tau||_{(VEnv \cup \{(x,\text{region}(\rho))\},\text{L} \cup \{(\rho,\text{R})\})} = ||\tau||_{(VEnv,L)}$.

This lemma states that the set of legal values corresponding to a "well formed type" τ, is independent of a region on which this type is not dependent. Hence that region can be safely deallocated if necessary.

Proof: If τ is of the form int or Unknown then it is trivially true. If τ is of the form handle(ρ'', τ'') then $C \vdash \tau$ only if $C \vdash \rho'' : \tau''$ (from SS23) and in turn $\rho'' \in$Dom(Δ) from SS20. Since $\rho \notin$Dom(Δ), $\rho'' \neq \rho$. Therefore, $||\tau||_{(VEnv \cup \{(x,\text{region}(\rho))\},\text{L} \cup \{(\rho,\tau')\})}$ = region(ρ'') = $||\tau||_{(VEnv,L)}$.

The most important case is when τ is a pointer type, i.e. τ is of the form $\tau'' * \rho''$. In this case, using SS20 and (SS22 or SS21), we get $\rho'' \neq \rho$ and hence $||\tau||_{(VEnv \cup \{(x,\text{region}(\rho))\},\text{L} \cup \{(\rho,\tau')\})}$ = $\{Uninit\} \cup$Dom(L[ρ''].RS) = $||\tau||_{(VEnv,L)}$.

We now state and prove the invariant preservation theorem for expressions.

THEOREM 1. *If $C \vdash e : \tau$ and $C \vdash_{env}$(VEnv, L) then either (VEnv, L, e) $\longrightarrow^*_{expr}$Error or (VEnv, L, e) $\longrightarrow^*_{expr}$ (VEnv', L', v) such that $v \in ||\tau||_{(VEnv',L')}$ and $C \vdash_{env}$ (VEnv', L')*

This theorem states that given a typing context C, if e is a well typed expression typing to τ then evaluation of e in a well formed environment either fails because of a run-time check failure or gives

R18 (VEnv $\cup \{(x, v)\}$, L, x) $\longrightarrow_{expr}$ (VEnv $\cup \{(x, v)\}$, L, v)

R19 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, E op E}_2\text{)} \longrightarrow_{expr} \text{(VEnv', L', E' op E}_2\text{)}}$

R20 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, } v \text{ op E)} \longrightarrow_{expr} \text{(VEnv', L', } v \text{ op E')}}$

R21 (VEnv, L, m op n) $\longrightarrow_{expr}$ (VEnv, L, m op$_{Int}$ n))

R22 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, load/loadc E)} \longrightarrow_{expr} \text{(VEnv', L', load E')}}$

R23 (VEnv, L, load/loadc v_1) $\longrightarrow_{expr}$ (VEnv, L, getvalue(L, v_1)) $\boxed{(v_1)! = Uninit}$
where
getvalue(L, v_1) := L[ρ].RS[v_1] if $\exists \rho \in$ Dom(L) s.t. $v_1 \in$ L[ρ].Dom(RS)
$Uninit$ else

R24 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, loadU/loadcU E, E}_2\text{)} \longrightarrow_{expr} \text{(VEnv', L', loadU/loadcU E', E}_2\text{)}}$

R25 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, loadU/loadcU } v_1\text{, E)} \longrightarrow_{expr} \text{(VEnv', L', loadU/loadcU } v_1\text{, E')}}$

R26 (VEnv, L, loadU region(ρ), v_1) $\longrightarrow_{expr}$ (VEnv, L, getvalue(L, v_1, 4) }]) $\boxed{(v_1, v_1 + 4] \in \text{Dom(L[}\rho\text{].RS)}}$
where
getvalue(L, v_1, 4) := combine(L[ρ].(RS[v_1]),L[ρ].(RS[$v_1 + 1$]),L[ρ].(RS[$v_1 + 2$]),L[ρ].(RS[$v_1 + 3$]))
if $\exists \rho \in$ Dom(L) s.t. $[v_1, v_1 + 3] \in$ L[ρ].Dom(RS)
$Uninit$ else
and combine(b1, b2, b3, b4) := (b1 << 24) || (b2 << 16) || (b3 << 8) || (b4).

R27 (VEnv, L, loadcU region(ρ), v_1) $\longrightarrow_{expr}$ (VEnv, L, getvalue(L, v_1) }])

R28 (VEnv, L, cast E to τ) $\longrightarrow_{expr}$ (VEnv, L, E)

R29 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, castintpointer E, E}_2\text{ to } \tau\text{)} \longrightarrow_{expr} \text{(VEnv', L', castintpointer E', E}_2\text{ to } \tau\text{)}}$

R30 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, castintpointer } v\text{, E to } \tau\text{)} \longrightarrow_{expr} \text{(VEnv', L', castintpointer } v\text{, E' to } \tau\text{)}}$

R31 (VEnv, L, castintpointer (region(ρ), v to τ) $\longrightarrow_{expr}$ (VEnv, L, v) $\boxed{v \in \text{Dom(L[}\rho\text{].RS)}}$

R32 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolalloc(E, E}_2\text{))} \longrightarrow_{expr} \text{(VEnv', L', poolalloc(E', E}_2\text{))}}$

R33 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolalloc(}v\text{, E))} \longrightarrow_{expr} \text{(VEnv', L', poolalloc(}v\text{, E'))}}$

R34 (VEnv, L $\cup$ { (ρ, $\{a\ F; RS\}$) }, poolalloc(region(ρ), 1)) $\longrightarrow_{expr}$ (VEnv, L $\cup$ { (ρ, $\{F; RS\}$) }, a)

R35 (VEnv, L $\cup$ { (ρ, $\{\phi; RS\}$) }, poolalloc(region(ρ), 1)) $\longrightarrow_{expr}$ (VEnv, L[$\rho \mapsto \{\phi; RS[a \mapsto Uninit]\}$], a)
where a is a new address obtained from system allocator, i.e. a $\in$ H. H becomes H - { a }.

R36 (VEnv, L $\cup$ { (ρ, $\{F; RS\}$) }, poolalloc(region(ρ), m)) $\longrightarrow_{expr}$ (VEnv, Initialize(L $\cup$ { (ρ, $\{F; RS\}$) }, $Uninit$, a,m), a) if (m != 1)
where a is a new address for the array obtained from system allocator and Initialize initializes each element of the array with $Uninit$. H becomes H - { a , a + 1, ... , a + m-1 }

R37 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, (E, \&(E}_1\text{)[E}_2\text{]))} \longrightarrow_{expr} \text{(VEnv', L', E', \&(E}_1\text{)[E}_2\text{]))}}$

R38 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, (}v\text{, \&(E)[E}_2\text{]))} \longrightarrow_{expr} \text{(VEnv', L', }v\text{, \&(E')[E}_2\text{]))}}$

R39 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, (}v\text{, \&(}v_1\text{)[E]))} \longrightarrow_{expr} \text{(VEnv', L', (}v\text{, \&(}v_1\text{)[E']))}}$

R40 (VEnv, L, (region(ρ), $\&v1[v2]$)) $\longrightarrow_{expr}$ (VEnv, L, $v1 + v2 * \texttt{sizeof}(\tau)$) $\boxed{(v1 + v2 * \texttt{sizeof}(\tau)) \in \text{Dom(L[}\rho\text{].RS)}}$
where τ is the "static" type of the individual element of the array, available from the declaration.
Note that sizeof(τ) is a compile time constant.

Figure 16. Operational Semantic Rules for expressions

Lemma R2* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, x = E) $\longrightarrow^*_{stmt}$ (VEnv', L', x = E').
Proof: By induction on the number of steps in the derivation of (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E').
Base case: zero steps. Trivially true.
Induction Hypothesis : True for 'k' steps in derivation.
For 'k+1' steps, we have
(VEnv, L, E) $\longrightarrow_{expr}$ ... $\longrightarrow_{expr}$ (k steps) (VEnv", L", E") $\longrightarrow_{expr}$ (VEnv', L', E').
Using induction hypothesis, we have (VEnv, L, x = E) $\longrightarrow^*_{stmt}$ (VEnv", L", x = E").
We also have (VEnv", L", E") $\longrightarrow_{expr}$ (VEnv', L', E')
Using R2, we have (VEnv", L", X = E") $\longrightarrow_{stmt}$ (VEnv', L', x = E'). q.e.d.

Lemma R1* If (VEnv, L, S1) $\longrightarrow^*_{stmt}$ (VEnv', L', S1') then (VEnv, L, S1 ; S2) $\longrightarrow^*_{stmt}$ (VEnv', L', S1' ; S2)

Lemma R4* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, store/storec E, E_2) $\longrightarrow^*_{stmt}$ (VEnv', L', store E', E_2).

Lemma R5* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, store/storec v,E) $\longrightarrow^*_{stmt}$ (VEnv', L', store v, E') .
If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, storeU/storecU E, E_2, E_3) $\longrightarrow^*_{stmt}$ (VEnv', L', storeU/storecU E', E_2, E_3) .

Lemma R8* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, storeU/storecU v_1, E, E_3) $\longrightarrow^*_{stmt}$ (VEnv', L', storeU/storecU v_1, E', E_3) .

Lemma R9* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, storeU/storecU v_1, v_2, E) $\longrightarrow^*_{stmt}$ (VEnv', L', storeU/storecU v_1, v_2, E') .

Lemma R12* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, poolfree(E, E_2) $\longrightarrow^*_{stmt}$ (VEnv', L', poolfree(E', E_2) .

Lemma R13* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, poolfree(v, E) $\longrightarrow^*_{stmt}$ (VEnv', L', poolfree(v, E') .

Lemma R16* If (VEnv, L, S) $\longrightarrow^*_{stmt}$ (VEnv', L', S') then (VEnv, L, pool{S}pop(ρ) $\longrightarrow^*_{stmt}$ (VEnv', L', pool{S'}pop(ρ)) .

Lemma R19* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, E op E_2) $\longrightarrow^*_{expr}$ (VEnv', L', E' op E_2) .

Lemma R20* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, v op E) $\longrightarrow^*_{expr}$ (VEnv', L', v op E') .

Lemma R22* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, load/loadc E) $\longrightarrow^*_{expr}$ (VEnv', L', load E') .

Lemma R24* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, loadU/loadcU E, E_2) $\longrightarrow^*_{expr}$ (VEnv', L', loadU/loadcU E', E_2) .

Lemma R25* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, loadU/loadcU v_1, E) $\longrightarrow^*_{expr}$ (VEnv', L', loadU/loadcU v_1, E') .

Lemma R29* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, castintpointer E, E_2 to τ) $\longrightarrow^*_{expr}$ (VEnv', L', castintpointer E', E_2 to τ) .

Lemma R30* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, castintpointer v, E to τ) $\longrightarrow^*_{expr}$ (VEnv', L', castintpointer v, E' to τ) .

Lemma R32* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, poolalloc(E, E_2)) $\longrightarrow^*_{expr}$ (VEnv', L', poolalloc(E', E_2)) .

Lemma R33* If (VEnv, L, E) $\longrightarrow^*_{expr}$ (VEnv', L', E') then (VEnv, L, poolalloc(v, E)) $\longrightarrow^*_{expr}$ (VEnv', L', poolalloc(v, E')) .

Lemma R37* If (VEnv, L, E) $\longrightarrow_{expr}$ (VEnv', L', E') then (VEnv, L, &(E, E_1)[E_2])) $\longrightarrow_{expr}$ (VEnv', L', &(E', E_1)[E_2])) .

Lemma R38* If (VEnv, L, E) $\longrightarrow_{expr}$ (VEnv', L', E') then (VEnv, L, &(v, E)[E_2])) $\longrightarrow_{expr}$ (VEnv', L', &(v, E')[E_2])) .

Lemma R39* If (VEnv, L, E) $\longrightarrow_{expr}$ (VEnv', L', E') then (VEnv, L, v, &(v_1)[E])) $\longrightarrow_{expr}$ (VEnv', L', &(v, v_1)[E'])) .

Figure 17. Lemmas for operational semantics

a value of the appropriate type along with another well formed environment.

Proof: The proof of this theorem is by induction on the structure of typing derivation of $C \vdash e : \tau$.

Based on the last rule used in the typing derivation of e, we have the following (exhaustive) list of cases:

- SS0
  e must be of the form x and $x \in$ Dom(Γ).
  Since $C \vdash_{env}$ (VEnv, L), from **Inv1** we get $x \in$ Dom(VEnv).
  Now consider (VEnv, L, x) with x ∈ Dom(VEnv). Rule **R18** applies. Hence, (VEnv, L, x) $\longrightarrow_{expr}$ (VEnv, L, v) where v is the image of x in VEnv. q.e.d.
- SS1
  e must be of the form n.
  Trivially, (VEnv, L, e) $\longrightarrow^*_{expr}$ (VEnv, L, n). q.e.d.
- SS2
  e must be of the form e_1 op e_2 with $C \vdash e_1$: `int` and $C \vdash e_2$: `int`.
  Using induction hypothesis, we have (VEnv, L, e_1) $\longrightarrow^*_{expr}$ Error or (VEnv", L", v_1) with $v_1 \in ||\texttt{int}||_{(VEnv'', L'')}$ and $C \vdash_{env}$(VEnv", L").
  If (VEnv, L, e_1) $\longrightarrow^*_{expr}$ Error then q.e.d.
  If not, using induction hypothesis again, we have (VEnv", L", e_2) $\longrightarrow_{expr}$ Error or (VEnv', L', v_2) with $v_2 \in ||\texttt{int}||_{(VEnv', L')}$ and $C \vdash_{env}$(VEnv', L').
  If (VEnv, L, e_2) $\longrightarrow^*_{expr}$ Error then q.e.d.
  If not,
  - Using lemma **R19***, we have (VEnv, L, e_1 op e_2) $\longrightarrow^*_{expr}$ (VEnv", L", v_1 op e_2)
  - Using lemma **R20***, we have (VEnv", L", v_1 op e_2) $\longrightarrow^*_{expr}$ (VEnv', L', v_1 op v_2)
  - Since v_1, $v_2 \in$ Int, using **R21** we have (VEnv', L', v_1 op$_{Int}$ v_2). with v_1 op$_{Int}$ $v_2 \in$ Int and ($C \vdash_{env}$(VEnv', L'). q.e.d.
- SS3
  e must be of the form $Uninit$.
  Trivially (VEnv, L, $Uninit$) $\longrightarrow^*_{expr}$ (VEnv, L, $Uninit$) and since $Uninit \in ||\tau||_{(VEnv, L)}$ where $\tau \neq$handle(ρ', τ') , q.e.d.

- SS4
  e must be of the form load e' and there exists ρ such that $C \vdash \rho : \tau$ and $C \vdash e' : \tau * \rho$ and $\tau \notin \{Unknown, \texttt{char}\}$.
  Now using induction hypothesis we have either (VEnv, L, e') $\longrightarrow^*_{expr}$ Error or (VEnv", L", v') such that $v' \in (\{Uninit\} \cup$ Dom(L"[ρ].RS)) and (VEnv", L") is well formed environment.
  If $v' = Uninit$ then by **R23**, (VEnv", L", load v') $\longrightarrow_{expr}$ Error and q.e.d.
  If $v' \in$ Dom(L"[ρ].RS) then from the load rule **R23**, we get (VEnv", L", load v') $\longrightarrow_{expr}$ (VEnv", L", getvalue(L", v')).
  From the **"getvalue" lemma** we get getvalue(L", v') $\in ||\tau||_{(VEnv'', L'')}$.
  Now using **R22*** we have the result. q.e.d.
  Informally, the proof step says that in case of load from type consistent memory, the address points to a correct object in the region or its uninitialized value, hence it progresses to an error or gives a value of the correct type.
- SS4char
  Similar to above.
- SS5
  e must be of the form loadU x, e_2, τ must be `int` with $C \vdash e_2 : \tau' * \rho$ and $C \vdash \tau : Unknown$ and $C \vdash x$:handle($\rho, Unknown$).
  Using induction hypothesis, we have either (VEnv, L, x) $\longrightarrow^*_{expr}$ Error or (VEnv", L", v_1) with v_1 = region(ρ) and $C \vdash_{env}$ (VEnv", L").
  If not Error, using lemma **R24***, we get (VEnv, L, loadU $e1, e_2$) $\longrightarrow^*_{expr}$ (VEnv" , L", loadU region(ρ), e_2).
  Again from induction hypothesis on e_2 we have (VEnv", L", e_2) $\longrightarrow^*_{expr}$ Error or (VEnv', L', v_2) with $v_2 \in \{Uninit\} \cup$ Dom(L[ρ].RS) and $C \vdash_{env}$ (VEnv', L').
  If (VEnv", L", e_2) $\longrightarrow^*_{expr}$ Error then q.e.d.
  If not, using lemma **R25***, we get (VEnv", L", load region(ρ), e_2) $\longrightarrow^*_{expr}$ (VEnv', L', load region(ρ), v_2) and $v_2 \in \{Uninit\} \cup$ Dom(L[ρ].RS). Now R26 applies and since we check that v_2 + 3 $\in$ Dom(L[ρ].RS), we get Error or (VEnv', L', getvalue(L, v_1, 4).
  From **"Getvalue" lemma2**, getvalue just retrieves the value from the location, and doesn't change any of the invariants of (VEnv', L'). q.e.d.
- SS5char
  Similar to SS4char.

- SS6
  e has to be of form poolalloc($x, e2$) and τ of the form $\tau' * \rho$ and $C \vdash \rho : \tau'$ and $C \vdash x$:handle(ρ, τ') and $C \vdash e2$: `int`
  Using induction hypothesis, (VEnv, L, x) $\longrightarrow_{expr}$ Error or (VEnv", L", v) s.t. $v =$region(ρ). If not error, (VEnv", L"', $e2$) $\longrightarrow_{expr}$ (VEnv"', L", n). If n is one then either rule (**R34** or rule **R35** applies.
  If rule **R34** applies, then the value returned is a value from the free list, and from invariant 4 for well-formed environments, we get that the value returned is of the correct type. Since we just removed an element from the freelist, the **Inv5** still holds and we continue to have a well formed environment.
  If **R35** applies, then we add an element to the set of addresses of this region and since the new address is from system heap H, **Inv6** continues to hold. **Inv4** holds in the new environment since we have initialized it with $Uninit$ value. **Inv5** holds since the free list has not changed. Hence the well formedness of the environment remains intact.
  The case where n is not one is similar by using the array allocation rule. q.e.d.
- SS7
  e has to be of the form castintpointer x, e'' to $\tau' * \rho$ with τ of the form $\tau' * \rho$ with $C \vdash \rho : \tau'$, $C \vdash x$:handle(ρ, τ') and $C \vdash e'' : int$
  Using induction hypothesis, (VEnv, L, x) $\longrightarrow^*_{expr}$ Error or (VEnv", L", v') such that v' = region(ρ). If not error, using lemma **R30*** we have (VEnv, L, castintpointer e', e" to $\tau' * \rho$) $\longrightarrow^*_{expr}$ (VEnv", L", castintpointer region(ρ), e'' to $\tau' * \rho$).
  Using induction hypothesis again (VEnv", L", e'') $\longrightarrow^*_{expr}$ Error or (VEnv', L', v'') such that $v'' \in$ Int. If not error, using lemma **R31*** we have (VEnv", L", castintpointer region(ρ), e'' to $\tau' * \rho$) $\longrightarrow^*_{expr}$ (VEnv", L", castintpointer region(ρ), v_2 to $\tau' * \rho$) with $v_2 \in$ Int.
  From rule **R31** from the operational semantics, we get (VEnv", L", castintpointer region(ρ), v_2 to $\tau' * \rho$) $\longrightarrow_{expr}$ Error or (VEnv", L", v_2) s.t. $v_2 \in$ Dom(L"[ρ].RS) or in otherwords $v_2 \in ||\tau' * \rho||_{(VEnv'', L'')}$. q.e.d.
- SS8
  This is straightforward application of the induction hypothesis and rule **R28**. None of the invariants change since the invariants for pointer types depend on the region attribute and not the actual type (See the discussion of SS8 in Section 10.1 to undestand how we can support casts between arbitrary pointer types).
- SS9
  All array accesses are checked for the pool boundaries and alignment. e is of the form $x, \&e2[e3]$ with $C \vdash \rho : \tau$, $C \vdash x$:handle(ρ, τ), $C \vdash e2 : \tau * \rho$, $C \vdash e3 : int$.
  Using induction hypothesis we get, (VEnv, L, x) $\longrightarrow^*_{expr}$ Error or (VEnv", L", region(ρ)) with $C \vdash_{env}$ (VEnv", L").
  If not error, then using lemma **R37*** we get (VEnv, L, $x, \&e2[e3]$) $\longrightarrow^*_{expr}$ (VEnv", L", region(ρ), $\&e2[e3]$).
  Using induction hypothesis again we get, (VEnv", L", $e2$) $\longrightarrow^*_{expr}$ Error or (VEnv"', L"', v_2) with $v_2 \in ||\tau * \rho||_{(VEnv''', L''')}$ and $C \vdash_{env}$ (VEnv"', L"').
  If not error, then using lemma **R38*** we get (VEnv", L", region(ρ), $\&e2[e3]$) $\longrightarrow^*_{expr}$ (VEnv"', L"', region(ρ), $\&v_2[e3]$).
  Using induction hypothesis again we get, (VEnv"', L"', $e3$) $\longrightarrow^*_{expr}$ Error or (VEnv', L', n) with $C \vdash_{env}$ (VEnv', L').
  If not error, then using lemma **R39*** we get (VEnv"', L"', region(ρ), $\&v_2[e3]$) $\longrightarrow^*_{expr}$ (VEnv', L', region(ρ), $\&v_2[m]$).
  Now **R40** applies. If ($v_2 + m*$sizeof(τ)) $\notin$ Dom(L'[ρ]) then Error else (VEnv', L', $v_2 + m*$sizeof(τ)) with ($v_2 + m*$sizeof(τ)) $\in ||\tau * \rho||_{(VEnv', L')}$ and $C \vdash_{env}$ (VEnv',L'). q.e.d.
- SS10
  This is straightforward application of the induction hypothesis and rule **R28**.

THEOREM 2. *If $C \vdash S$ and $C \vdash_{env}$(VEnv, L) then either (VEnv, L, S) $\longrightarrow^*_{stmt}$Error or (VEnv, L, S) $\longrightarrow^*_{stmt}$ (VEnv', L', ϵ) and $C \vdash_{env}$ (VEnv', L').*

Here $\longrightarrow^*_{stmt}$ represents reflexive transitive closure of $\longrightarrow_{stmt}$. This theorem states that given a typing environment C, if statement S is well typed in that typing environment then evaluation of S either fails because of a run-time check failure or terminates along with another well formed environment. To put it differently, it does not get stuck because of type violation (e.g. trying to access non-existent memory location).

Proof: The proof of this theorem is by induction on the structure of typing derivation of $C \vdash S$.

Based on the last rule used in the typing derivation of S, we have the following (exhaustive) list of cases:

- SS11
  S is of the form ϵ.
  Trivial case.

- SS12
  S is of the form S1; S2 with $C \vdash S1$ and $C \vdash S2$.
  Using induction hypothesis, (VEnv, L, S1) $\longrightarrow^*_{stmt}$ Error or (VEnv", L", ϵ).
  If not Error, Using R1* (VEnv, L, S1 ; S2) $\longrightarrow^*_{stmt}$ (VEnv", L", S2)
  Using induction hypothesis again, we have (VEnv", L", S2) $\longrightarrow^*_{stmt}$ Error or (VEnv", L", ϵ). Using transitivity of $\longrightarrow^*_{stmt}$ q.e.d.
- SS13
  S is of the form x = e with $C \vdash x : \tau$ and $C \vdash e : \tau$.
  Using Theorem 1 on $C \vdash e : \tau$, we have (VEnv, L, e) $\longrightarrow^*_{expr}$ Error or (VEnv", L", v) with $v \in ||\tau * \rho||_{(VEnv'',L'')}$ and $C \vdash_{env}$ (VEnv", L").
  Using lemma **R2***, we get (VEnv, L, x = e) $\longrightarrow^*_{stmt}$ (VEnv", L", x = v). with $v \in ||\tau||_{(VEnv'',L'')}$.
  Using R3 we get (VEnv", L", x = v) $\longrightarrow^*_{stmt}$ (VEnv"[x $\mapsto v$], L", ϵ).
  Let VEnv' = VEnv"[x $\mapsto v$]. Now we need to prove that $C \vdash_{env}$ (VEnv', L"). **Inv1, Inv2, Inv4, Inv5, Inv6** can be trivially proved from $C \vdash_{env}$ (VEnv", L").
  We have $C \vdash x : \tau$ and $v \in ||\tau||_{(VEnv'',L'')}$. So $v \in ||\tau||_{(VEnv',L')}$ since VEnv' differs from VEnv" differs only in the mapping of x. We also have VEnv'[x] = v. So **Inv3** continues to hold. Hence $C \vdash_{env}$ (VEnv', L").
- SS14
  S if of the form $C \vdash$ store e_2, e_1, with $C \vdash \rho : \tau$, $C \vdash e_1 : \tau * \rho$, and $C \vdash e_2 : \tau$.
  Using Theorem 1, we have (VEnv, L, e_2) $\longrightarrow^*_{expr}$ Error or (VEnv", L", v_2) with $v_2 \in ||\tau||_{(VEnv'',L'')}$ and $C \vdash_{env}$ (VEnv", L").
  If not error, Using **R4***, we have (VEnv, L, store e_2, e_1) $\longrightarrow^*_{stmt}$ (VEnv", L", store v_2, e_1).
  First from **Inv2** we have $\rho \in$ Dom(L).
  Now using theorem 1, we have (VEnv", L", e_1) $\longrightarrow^*_{expr}$ Error or (VEnv', L', v_1) with $v_1 \in ||\tau * \rho||_{(VEnv',L')}$ and $C \vdash_{env}$ (VEnv', L').
  If not error, Using **R5***, we have (VEnv", L", store v_2, e_1) $\longrightarrow^*_{stmt}$ (VEnv', L', store v_2, v_1).
  If v_1 is $Uninit$ then using **R6** we get (VEnv', L', store v_2, v_1) $\longrightarrow_{stmt}$ Error and q.e.d.
  If not,(VEnv', L', store v_2, v_1) $\longrightarrow_{stmt}$ (VEnv', update(L', v_1, v_2), ϵ from **R6**.
  Using the **Update lemma1**, $C \vdash_{env}$ (VEnv', update(L', v_1, v_2). Similarly we can prove for the SS14char case.
- SS15 and SS15char
  same as above
- SS16
  S is of the form poolfree(x, e_2) and $C \vdash \rho : \tau$ and $C \vdash x :$ handle(ρ, τ) and $C \vdash e_2 : \tau * \rho$.
  Using **Theorem1**, we get (VEnv, L, x) (VEnv", L", v_1) s.t. $v_1 \in ||$handle$(\rho, \tau)||_{(VEnv'',L'')}$ and $C \vdash_{env}$ (VEnv", L").
  Now using **Theorem 1** again, we get (VEnv", L", e_2) $\longrightarrow^*_{expr}$ Error or (VEnv', L', v_2) with $v_2 \in ||\tau * \rho||_{(VEnv',L')}$ and $C \vdash_{env}$ (VEnv', L').
  Using **R12*** and **R13***, we get (VEnv, L, poolfree(x, e_2)) $\longrightarrow^*_{stmt}$ (VEnv', L', poolfree(v_1, v_2)).
  From the definition of $||\tau||$, v_1 = region(ρ) and $v_2 \in \{Uninit\} \cup$Dom(L'[ρ].RS).
  If (v_2 == $Uninit$) then from **R14**, (VEnv', L', poolfree(v_1, v_2)) $\longrightarrow^*_{stmt}$ Error.
  If not, from **Inv2** $\rho \in$ Dom(L') and **R14** applies. Let L' = L" $\cup \{(\rho, \{F; RS\})\}$, $\rho \notin$Dom(L"). So (VEnv', L" $\cup \{(\rho, \{F; RS\})\}$, poolfree(region($\rho$), v_2)) $\longrightarrow_{stmt}$ (VEnv', L" $\cup \{(\rho, \{v_2 F; RS\})\}, \epsilon$). We just need to prove that $C \vdash_{env}$ (VEnv', L" $\cup \{(\rho, \{v_2 F; RS\})\}$).
  We already have $C \vdash_{env}$ (VEnv', L'). So for (VEnv', L" $\cup \{(\rho, \{v_2 F; RS\})\}$) **Inv1, Inv2, Inv3** trivially hold as they are the same for (VEnv', L'). **Inv4** holds since $\forall \rho$ L'[ρ].RS is unmodified. **Inv5** holds since $v_2 \in$ Dom(L'[ρ].RS). **Inv6** holds since $\forall \rho$ L'[ρ].RS and H is unmodified. Hence $C \vdash_{env}$ (VEnv', L" $\cup \{(\rho, \{v_2 F; RS\})\}$). q.e.d.
- SS17
  S is of the form poolinit(ρ, τ) x { S' } with $C(= \Gamma, \Delta) \vdash \tau$ and $\Gamma[x \mapsto$handle$(\rho, \tau)], \Delta[\rho \mapsto \tau] \vdash S'$ with $x \notin$Dom(Γ) and $\rho \notin$Dom(Δ).
  Rule **R15** applies if $\rho \notin$ Dom(L). We already have $\rho \notin$ Dom(Δ) and from **Inv2** we have $\rho \notin$ Dom(L). So **R15** applies. Hence, (VEnv, L, poolinit(ρ, τ)x { S' }) $\longrightarrow_{stmt}$ (VEnv $\cup \{(x,$region$(\rho))\}$, L $\cup \{(\rho, \{\phi; \phi\})\}$, pool { S' }pop($\rho$)).
  Let VEnv" = VEnv $\cup \{(x,$region$(\rho))\}$ and L" = L $\cup \{(\rho, \{\phi; \phi\})\}$.
  Let C' (Γ', Δ') = $\Gamma[x \mapsto$handle$(\rho, \tau)], \Delta[\rho \mapsto \tau]$. We first need to prove that $C' \vdash_{env}$ (VEnv", L").
  - **Inv1**
    From **Inv1** of (VEnv, L), we get Dom(Γ) = Dom(VEnv).
    Dom(Γ') = Dom(Γ) $\cup$ { x }
    = Dom(VEnv) $\cup$ { x }
    = Dom(VEnv")
    So **Inv1** holds.
  - **Inv2**
    From **Inv2** of (VEnv, L), we get Dom(Δ) = Dom(L).
    Dom(Δ') = Dom(Δ) $\cup$ { ρ }
    = Dom(L) $\cup$ { ρ }
    = Dom(L")
    So **Inv2** holds.
  - **Inv3**
    From **Inv3** of (VEnv, L) we get $\forall y \in$ Dom(VEnv), if $C \vdash y : \tau'$ then VEnv[y] $\in ||\tau'||_{(VEnv,L)}$
    Since Dom($VEnv'$) = Dom(VEnv) $\cup$ { x }. We just need to prove the invariant for x. We have $C' \vdash x$: handle(ρ, τ) and VEnv"[x] = region(ρ), and so the invariant **Inv3** continues to hold.
  - **Inv4**
    From **Inv4** of (VEnv, L) we get $\forall \rho' \in$ Dom(L), if $C \vdash \rho' : \tau'$ then $\forall v \in$ Dom((L[ρ']).RS), L[ρ'].RS[v] $\in ||\tau'||_{(VEnv,L)}$.
    $\forall \rho' \in$ Dom(L"),
    - if $\rho' \in$Dom(L), then **Inv4** continues to hold because none of the previous regions changed.
    - if $\rho' = \rho$, then since Dom(L"[ρ].RS) = ϕ the invariant holds trivially.

    Hence **Inv4** holds.
  - **Inv5**
    From **Inv5** of (VEnv, L) we get $\forall \rho' \in$ Dom(L), Dom((L[ρ']).F) $\subseteq$ Dom((L[ρ']).RS).
    $\forall \rho' \in$ Dom(L"),
    - if $\rho' \in$Dom(L), then **Inv5** continues to hold because none of the regions in L are changed.
    - if $\rho' = \rho$, then since Dom(L"[ρ].RS) = Dom(L"[ρ].F) = ϕ, the invariant holds trivially.

    Hence **Inv5** holds.
  - **Inv6**
    From **Inv6** of (VEnv, L) we get $\forall \rho_1 \rho_2 \in$ Dom(L), if $\rho_1 \neq \rho_2$ then Dom((L[ρ_1]).RS) $\cap$ Dom((L[ρ_2]).RS) = ϕ.
    $\forall \rho_1 \rho_2 \in$ Dom(L"),
    - if $\rho_1, \rho_2 \in$Dom(L), then **Inv6** continues to hold because none of the regions in L changed.

– if either of ρ_1 or ρ_2 = ρ, then since Dom(L"[ρ].RS) = ϕ, the invariant holds trivially.

Moreover, H does not change so the **Inv6** holds.

Hence $C' \vdash_{env}$ (VEnv", L").
Now using induction hypothesis, we get (VEnv", L", S) $\longrightarrow^*_{stmt}$ (VEnv"', L"', ϵ) and $C' \vdash_{env}$ (VEnv"', L"').
Using lemma **R16*** we get (VEnv", L", pool { S }pop(ρ)) $\longrightarrow^*_{stmt}$ (VEnv"', L"', pool { ϵ }pop(ρ)).
Now $C' \vdash_{env}$ (VEnv"', L"').
So from **Inv1** we have $x \in$Dom(VEnv"').
From **Inv3** we have VEnv"'[x] =region(ρ).
From **Inv2** we have ρ in Dom(L"').

Let VEnv"' = VEnv' $\cup \{(x$,region(ρ))$\}$ and L"' = L' $\cup \{(\rho$,R)$\}$.

Now rule **R17** applies.
(VEnv"', L"', pool { ϵ }pop(ρ)) $\longrightarrow_{stmt}$ (VEnv', L', ϵ).
Let H"' be the set of unused system addresses before this operation. and H' be the set of unsed system addresses after.
Then H' = H"' $\cup$ Dom(L"'[ρ].RS).
We need to prove that $C \vdash_{env}$ (VEnv', L').

Note that Dom(VEnv"') = Dom(VEnv') $\cup \{x\}$ where $x \notin$ Dom(VEnv').

- **Inv1**
  From **Inv1** of (VEnv"', L"'), we get Dom(Γ') = Dom(VEnv"').
  So Dom(Γ) $\cup\{x\}$ =Dom(VEnv') $\cup \{x\}$. but $x \notin$Dom(Γ) and $x \notin$Dom(VEnv'). So Dom(Γ) = Dom(VEnv') . q.e.d.
- **Inv2**
  From **Inv2** of (VEnv"', L"'), we get Dom(Δ') = Dom(L"').
  So Dom(Δ) $\cup\{\rho\}$ =Dom(L') $\cup \{\rho\}$. but $\rho \notin$Dom(Δ) and $\rho \notin$Dom(L'). So Dom(Δ) = Dom(L') . q.e.d.
- **Inv3**
  From **Inv3** of (VEnv"', L"'), we get $\forall y \in$Dom(VEnv"'), if $C' \vdash y : \tau$ then VEnv"'[y] $\in ||\tau||_{(VEnv''',L''')}$.
  $\forall y \in$Dom(VEnv'), if $C \vdash y : \tau$ then we have $y \neq x$ and $y \in$Dom(VEnv"').
  VEnv"'[y] $\in ||\tau||_{(VEnv''',L''')}$. Since $y \neq x$, VEnv"'[y] = VEnv'[y]. From **Well formed type lemma** and **Safe region deallocate lemma** we have $||\tau||_{(VEnv''',L''')} = ||\tau||_{(VEnv',L')}$.
  q.e.d.
- **Inv4**
  From **Inv4** of (VEnv"', L"') we get $\forall \rho' \in$ Dom(L"'), if $C' \vdash \rho' : \tau'$ then $\forall v \in$ Dom((L"'[ρ']).RS), L"'[ρ'].RS[v] $\in ||\tau'||_{(VEnv''',L''')}$.
  $\forall \rho' \in$ Dom(L'), if $C \vdash \rho' : \tau'$ then we know that $\rho' \neq \rho$ and $\rho' \in$Dom(L"').
  So we have, $\forall v \in$ Dom((L"'[ρ']).RS), L"'[ρ'].RS[v] $\in ||\tau'||_{(VEnv''',L''')}$.
  But $\rho' \neq \rho$ so $\forall v \in$ Dom((L'[ρ']).RS), L'[ρ'].RS[v] $\in ||\tau'||_{(VEnv''',L''')}$.
  Now using **Well formed type lemma** and **Safe region deallocate lemma** we have $||\tau'||_{(VEnv''',L''')} = ||\tau||_{(VEnv',L')}$.
  q.e.d.
- **Inv5**
  From **Inv5** of (VEnv"', L"') we get $\forall \rho' \in$ Dom(L"'), Dom((L"'[ρ']).F) $\subseteq$ Dom((L"'[ρ']).RS).
  This trivially holds for (VEnv', L') since we have just taken element out of the map for L"'.
- **Inv6**
  From **Inv6** of (VEnv"', L"') we get $\forall \rho_1\rho_2 \in$ Dom(L"'), if $\rho_1 \neq \rho_2$ then Dom((L"'[ρ_1]).RS) $\cap$ Dom(L"'[ρ_2].RS) = ϕ.
  This trivially holds for (VEnv', L') since we have just taken an element out of the map for L"'.
  Also, $\forall \rho' \in$ Dom(L'), $\rho' \neq \rho$ and Dom(L'[ρ'].RS) = Dom(L"'[ρ'].RS). From **Inv6** of (VEnv"',L"') Dom(L"'[ρ'].RS) $\cap$ H"' = ϕ and Dom(L"'[ρ'].RS) $\cap$ Dom(L"'[ρ].RS) = ϕ. So, Dom(L'[ρ'].RS) $\cap$ (H"' $\cup$ Dom(L"'[ρ].RS)) = ϕ.
  Hence **Inv6** holds.

q.e.d.