YARRA: An Extension to C for Data Integrity and Partial Safety

(MSR-TR-2010-158)

Cole Schlesinger
Princeton University
cschlesi@cs.princeton.edu

Karthik Pattabiraman
University of British Columbia
karthikp@ece.ubc.ca

Nikhil Swamy
Microsoft Research
nswamy@microsoft.com

David Walker
Princeton University
dpw@cs.princeton.edu

Ben Zorn
Microsoft Research
zorn@microsoft.com

Abstract

Modern applications contain libraries and components written by different people at different times in different languages, often including unsafe languages like C or C++. As a result, one bug, such as a buffer overflow, in any component, can compromise the security and reliability of every other component. To help mitigate these problems, we introduce YARRA, a conservative extension to C with mechanisms for enforcing data integrity and partial safety, even when code is linked against unknown C libraries or binaries. YARRA programmers specify their data integrity requirements by declaring *critical data types* and ascribing these critical types to important data structures. YARRA guarantees that such critical data is only written through pointers with the given static type. Any attempt to write to critical data through a pointer with an invalid type (perhaps because of a buffer overrun) is detected dynamically.

We formalize YARRA's semantics and prove the soundness of a program logic designed for use with the language. A key contribution is to show that YARRA's semantics are strong enough to support sound local reasoning and the use of a frame rule, even across calls to unknown, unverified code. We also demonstrate that YARRA's semantics can be implemented in several different ways, with different performance and pragmatic tradeoffs. In one implementation, we perform a source-to-source program transformation to ensure correct execution. In a second implementation, we do not rely upon having access to the entire source code, but instead use conventional hardware permissions to protect critical data. We evaluate our implementations using SPEC benchmarks to understand their performance. Additionally, we apply YARRA to four common applications with known non-control data vulnerabilities. We are able to use YARRA to defend against these attacks while sustaining a negligible impact on their end-to-end performance.

1. Introduction

Despite the widespread use of type-safe languages, most important applications contain components written in unsafe languages such as C and C++. Indeed, even the implementations of safe programming languages generally rely on libraries written in unsafe languages. In such multi-component systems, just one bug, such as a buffer overflow, in any unsafe component, can compromise the security and reliability of every other component.

Over the years, there has been a great deal of research aimed at making these unsafe systems more secure and reliable. In particular, much attention has been paid to preventing *control-based attacks* on unsafe programs. In this class of attack, the attacker attempts to use a program error, such as a buffer overflow or use-after-free, to overwrite control-data such as return addresses and function pointers to modify the control-flow of the program. Fortunately, commercially implemented mitigation techniques, including protecting the return address on the stack [6], turning off code execution in the heap [7], randomizing the layout of code and data in memory [26], and enforcing control-flow integrity [1], have reduced the effectiveness of control-flow attacks. Unfortunately, attackers are now looking to try new methods of attack.

In 2005, Chen *et al.* [5] illustrated that a variety of *non-control data attacks* can be launched against servers that implement HTTP, FTP, SSH, and other important applications. These attacks do not modify the control-flow of programs, but instead corrupt user identity data, configuration data, user input data or decision-making data to achieve the attacker's ends. Since 2005, due to the mitigations against control-based attacks, the prevalence of non-control data attacks has increased [22].

To help protect C programs against non-control data attacks, and, more generally, to improve the security and reliability of unsafe systems, we introduce YARRA, a new, conservative, lightweight extension to C. In YARRA, programmers introduce special type declarations and ascribe the special types to their *critical data structures*—those data structures upon which system reliability or security most depends. We call the special types *critical data types*.

Critical data types help programmers specify an intended *data integrity policy*. Programmers further specify their data integrity intentions by choosing, in any given program expression, *to use* a pointer with a critical type or *not to use* a pointer with a critical type. When accessing data through a pointer with a (static) critical type, a programmer declares that he or she expects the underlying memory to have that same critical type dynamically. When writing through a pointer that, statically, does not have a critical type, the programmer declares that he or she does not expect to be modifying memory with the (dynamic) critical type.

The concepts and properties mentioned above are intuitive and easy to use. They help defend against unintended modification of critical data, which is at the heart of all non-control data attacks, and moreover, is a general source of unreliability in C programs. More generally still, we argue that these properties are fundamental pillars upon which almost any local, modular reasoning about imperative programs depends. Of course, since these properties do not generally hold in C due to casting, buffer overruns, use-after-free and other unsafe operations, it is up to the YARRA implementation to enforce them.

Hence, this paper makes several contributions, including (1) the design of a new language extension, (2) the formalization of its semantics and reasoning principles, (3) the implementation of a compiler and runtime system, and (4) a demonstration of its effectiveness on a collection of applications. We discuss each of these contributions below.

Language design. YARRA is a lightweight, conservative extension to C with a number of properties that make it complementary to existing safety and security mechanisms. First, it allows programmers to use simple types to specify data integrity policies. Type declarations may be added to an existing code base, literally one at a time, incrementally hardening a program against non-control data attacks. Second, YARRA is designed to interoperate with libraries for which the source code is unavailable. Perhaps the source is owned by a third party, the library is written in a different language or the code is only available as a binary. In all of these cases, YARRA still provides protections for critical data, even if the library code is unsafe and contains buffer overruns. In other words, YARRA delivers a unique kind of *partial safety* to its users. In contrast, systems such as Cyclone [10], CCured [18], Softbound [16] and others that rely upon conventional array-bounds checking generally do not provide any guarantees whatsoever when there are buffer overruns in unchecked libraries. Despite this limitation, array-bounds checking remains a very useful technique, and we believe that YARRA provides protections that are both orthogonal and complementary.

Formal semantics. We provide an operational semantics and a sound program logic for a core model of YARRA. The program logic defines the formal or informal reasoning principles that programmers may use when analyzing their YARRA programs. A unique aspect of our program logic is its *type-based frame rule* that facilitates local reasoning about a security sensitive (or other) module. Specifically, we show (1) how to characterize any unprocessed, possibly "unsafe," external library call using a Hoare triple, (2) how to associate strong logical invariants with critical data types, like one would do in a conventional safe, strongly typed language, and (3) how to preserve such strong invariants across calls to external libraries using our frame rule.

Implementation of a compiler and runtime system. The semantics of YARRA may be implemented in more than one way. Different implementations have different performance tradeoffs in terms of time and space and different requirements in terms of access to source code for transformation. We currently have two implementations based on different techniques. The first technique, inspired by previous work on Write Integrity Testing (WIT) [2], instruments source code with dynamic checks that cannot be proven unnecessary at compile time. These checks implement a form of *inverse array bounds* checking.[1] In other words, rather than checking that a write is within the bounds of a particular object, the checks guarantee that it is *not* within the bounds of other, critical objects. While tantalizingly close to array bounds checking, the implementation and engineering tradeoffs involved are different. The second technique uses standard hardware protections provided by the virtual memory system. This technique, inspired by previous work on Samurai [20], makes copies of critical objects on separate pages. Prior to invoking untrusted library code, the implementation turns off hardware write permissions on the designated pages, thereby preventing unsafe libraries from corrupting critical data.

Applications of YARRA**.** We illustrate the effectiveness of YARRA on a collection of different applications drawn primarily from two domains: (1) applications with security-sensitive data that may be vulnerable to non-control data attacks, and (2) memory managers that maintain both client-inaccessible, internal data structures and client-accessible ordinary data. The latter is a stand-in for any general-purpose module with internal state that, for reliability (as opposed to security) purposes, would like to protect that state. We show that YARRA can effectively protect security critical data from known exploits with an overhead in execution time of 7–67% in the hardened module alone, and negligible overhead relative to the end-to-end performance of the application as a whole. The programmer integration effort was on the order of a few hundred modified lines of code in applications tens of thousands of lines long.

Limitations: While YARRA may be used to harden C programs against non-control data attacks, there are many vulnerabilities that YARRA does not defend against. In particular:

- YARRA does not protect against control-flow attacks, guard against stack smashing or provide any kind of control-flow integrity. Techniques such as the /GS flag, control-flow integrity [1] and/or DEP hardware protections should be used with YARRA if such attacks are considered a hazard.
- While YARRA protects against buggy libraries, it provides no protection against explicitly malicious library code. Our implementation does not currently prevent the untrusted library from making system calls that turn off hardware protection bits (or performing other arbitrarily malicious actions).

Further, neither YARRA implementation currently operates in multi-threaded contexts.

2. YARRA By Example

Background. A non-control data attack occurs when security-critical data allocated on the heap is unexpectedly modified. The display below shows an example of code vulnerable to such an attack. This example is drawn from Akritidis *et al.* [2] and was inspired by a true `nullhttpd` attack.

Code vulnerable to a non-control data attack

```
1  static char cgiCmd[1024];
2  static char cgiDir[1024];
3  void ProcessCGIRequest(char* msg, int sz) {
4     int flag, i=0;
5     while (i < sz) {
6          cgiCmd[i] = msg[i]; //buffer overrun here could overwrite cgiDir
7          i++;
8     }
9    flag = CheckRequest(cgiCmd); //input sanitization
10   if (flag) {
11      Log("..."); //buggy library could invalidate sanitization
12      ExecuteRequest(cgiDir, cgiCmd);
13   }}
```

In this example, a request (msg) is copied into a new buffer called cgiCmd. Next, a routine called CheckRequest checks that the command does not contain "..", which would allow an attacker to navigate out of the designated directory and execute any program, anywhere in the system. Finally, Log logs the request for future audits and ExecuteRequest concatenates the command to the designated directory path and executes it. Unfortunately, the routine is vulnerable when sz is larger than 1024. In this case, the copying operation overflows from cgiCmd into cgiDir, allowing an attacker to effectively execute any command in any directory on the user system. An additional concern is a potential time-of-check to time-of-use discrepancy in the code, that can be exploited, if, for example, the call to Log has a buffer overflow that allows cgiCmd to be overwritten after CheckRequest has been executed. Both of these vulnerabilites lead to non-control data attacks because they do not change the control flow of the C program. Hence, they will not be detected by mechanisms that check for control flow integrity.

There are two perspectives on this kind of attack:

[1] This implementation inspired the name YARRA (the inverse of ARRAY).

- *The conventional array-bounds perspective*: The fault lies with the write operations at line 7 and within the implementation of Log, since they misimplement indexing operations.
- *The data integrity perspective*: The fault lies in the definition and implementation of the cgiDir and cgiCmd data structures, since they fail to protect themselves from external agents.

These two different perspectives lead to different solutions with different engineering considerations. The conventional perspective, taken by systems such as SoftBound [16], leads one to maintain bounds on all data structures and to rewrite the code for every data access. Consequently, it cannot be applied when library source code is unavailable, *e.g.,* if a function like Log were to make library calls. In such a situation, all bets are off—a single missed bounds check may corrupt any data structure, anywhere in the program. In contrast, the data integrity perspective leads one to maintain bounds only for the high integrity (critical) data structures and indexing operations must be proven *not within* the bounds of these structures. This alternative perspective leads to a different set of implementation possibilities. For example, one may use conventional hardware protections to prevent writes to critical data, while still allowing safe linking with unmodified, possibly buggy libraries. We adopt the latter perspective in YARRA and show how it can be used to harden code against non-control data attacks.

2.1 Hardening `nullhttpd` with YARRA

The main new abstraction that YARRA provides is the *critical data type*. Critical data types have the rather unremarkable property that access to such data may only occur through a pointer with a corresponding (static) type. Working with critical data types demands a certain discipline. First, programmers must declare a critical type X. Having done so, programmers can designate (or *bless*) portions of memory as containing X objects and, as a result, they obtain X-typed references. X-typed memory should only be accessed using X-typed references. In return, YARRA ensures that the portions of memory that hold X-typed objects will never be corrupted by writes via untyped pointers, or by the effects of library code. When finished with an X object, a programmer can *unbless* a reference, undoing the protections on the referenced memory.

Programming with critical data types. The listing below shows how our example fragment from `nullhttpd` may be rewritten using YARRA's critical data types to foil both non-control data attacks. On line 1, we introduce a new critical data type, cchar, using a declaration much like C's typical declaration for structures. The type cchar is a new YARRA structure containing a single character field named cc. The type dchar (line 2) is another critical type, also with a single character field dc. At line 3, we declare that every element of cgiCmd is a cchar, meaning it can only be written by cchar pointers. Likewise, with cgiDir and dchar, at line 4. Finally, we modify line 8, to access the cc field of the YARRA structure, thereby indicating our *clear intention* to write to protected data.

YARRA's promise to programmers is that writes via non-critical pointers to memory locations holding critical objects will always be detected. Because the types cchar and dchar are unknown to the Log function and any library it may call, the functions use only non-critical pointers, and hence YARRA guarantees that both cgiDir and cgiCmd are uncorrupted at the call to ExecuteRequest. Further, at line 8, if there is a buffer overrun from cgiCmd into cgiDir, YARRA detects the error because a pointer with static type cchar* attempts to write to memory with (dynamic) YARRA type dchar. This illustrates the importance of using different YARRA types for logically distinct data structures. If one were to use the same type (say, cdchar) for both cgiCmd and cgiDir then YARRA would not prevent buffer overrun at line 8. In other words, structures that share the same type are not protected from each other; they are only protected from structures with other types.

Using critical data types in `nullhttpd`

```
1  yarra struct {char cc;} cchar;
2  yarra struct {char dc;} dchar;
3  static cchar cgiCmd[1024];
4  static dchar cgiDir[1024];
5  void ProcessCGIRequest(char* msg, int sz) {
6    int flag, i=0;
7    while (i < sz) {
8          cgiCmd[i].cc = msg[i]; //Yarra: cgiDir cannot be modified
9          i++;
10   }
11   flag = CheckRequest(cgiCmd);
12   if (flag) {
13     Log("..."); //Yarra: corruption of cgiDir, cgiCmd detected
14     ExecuteRequest(cgiDir, cgiCmd);
15   }}
```

Implementing* YARRA *protections. There are many ways to implement the protections YARRA offers—our current implementation offers two modes. In its *source protection* mode (inspired by WIT), our compiler uses the statically declared type of pointers to instrument memory accesses with suitable checks. For example, writes using non-critical pointers to locations are checked at run time to ensure they actually contain non-critical data. If they contain critical data, the program will abort. In its targeted *library protection* mode (inspired by Samurai), more suitable for situations in which code cannot be instrumented with checks (e.g., when linking with third-party binaries), we maintain backing stores for critical objects on separate pages. Prior to invoking potentially buggy library code, we turn off hardware write permissions on these pages to preserve their integrity. Writes from untyped pointers to critical objects proceed without failure, but, these writes only modify one copy of the object, leaving the version in the backing store unchanged. In contrast, writes to critical objects using well-typed references update both copies of the object. When a critical object is read using a well-typed pointer, checks inserted by our compiler ensure that the versions of the object in the main heap and the backing store are identical, thus detecting potential corruptions.

Reasoning about* YARRA *programs. Regardless of the implementation chosen, with both cgiCmd and cgiDir protected by YARRA, our semantics provides the programmer with powerful, sound, local reasoning principles. Any invariant over the objects cgiCmd and cgiDir is preserved across the call to the Log function, since Log is unable to modify critical memory locations. Additionally, an invariant on cgiDir (e.g., that cgiDir does not start with "..") is preserved across line 8, since YARRA ensures that the write to cgiCmd never modifies a dchar object. We formalize this principle in Section 3 in terms of a type-based *frame rule* and prove it sound.

2.2 Critical data and dynamic allocation

Our first example illustrated a simple use case for YARRA in which a set of memory locations have a single YARRA type for their entire lifetime. However, in order to handle dynamically allocated data structures, or memory that is reused for different purposes, we need a way to cast memory from one critical type to another.

In YARRA, memory pointed to by p is dynamically cast to a critical type T using the operation bless⟨T⟩(p) and cast back using unbless⟨T⟩(p). It is an error to attempt to bless memory protected at type T′ to another type T, unless T′ is a declared substructure of T[2]. Likewise, it is an error to attempt to unbless memory from type T when that memory location had not previously been blessed at

[2] An illegal cast of this sort might invalidate the protections supplied by T′.

```
1  yarra struct {int tag;} metaT;
2  yarra struct {int junk;} unusedT;
3  union item {
4    unusedT unused;
5    int used;
6  };
7  static metaT meta[SIZE];
8  static item data[SIZE];
9  int *alloc() {
10   int i;
11   for (i=0; i<SIZE; i++) {
12     if (meta[i].tag == 0) {
13       meta[i].tag = 1;
14       unbless⟨unusedT⟩(&data[i].unused);
15       return data+i;
16   } }
17   abort("out of memory");
18 }
19 void free(int *datum) {
20   if (datum >= data && datum < data+SIZE) {
21     int i = datum - data;
22     if (meta[i].tag == 1) {
23       if (vacant⟨unusedT⟩(&data[i])) {
24         meta[i].tag = 0;
25         bless⟨unusedT⟩(&data[i].unused);
26         return;
27   } } }
28   abort("client error"); }
```

Figure 1. A simplified memory manager

T. These sorts of errors are detected at runtime by the instrumentation inserted by our compiler. YARRA also provides the operation isIn⟨T⟩(p), which returns true if p dynamically has type T and false if it does not. If p points to memory which has been blessed at type T but which has been corrupted by a write via an untyped pointer, YARRA causes the program to abort—this situation can be detected, if, for example, the two copies of the T-object in question are not synchronized. Finally, YARRA provides the command vacant⟨T⟩(p), which returns true if p points to completely unprotected memory of size **sizeof**(T) and false otherwise.

Figure 1 shows a simple memory allocator that uses bless and unbless to protect its metadata, hence increasing its reliability, even when linked against buggy clients. While the allocator shown is extremely simple, we have used the same principles to protect BGET [25], a standard, publicly available allocator for C.

The allocator relies on a few simple invariants (where i ranges from 0 to SIZE−1): (1) the elements i of the meta array have critical type metaT, preventing a buggy client program from modifying allocator meta data; (2) the meta array contains integers that are either 0 or 1; (3) if meta[i] is 0 then data[i] is not allocated and dynamically has critical type unusedT, preventing a client from using it; and (4) if meta[i] is 1 then data[i] is allocated and dynamically does not have critical type unusedT, allowing a client to use it as needed.

Given these invariants, consider the effects of the alloc and free routines. In alloc, the code searches for a free cell (one with meta[i].tag == 0), assigns the meta[i] tag to 1 (allocated state), and unblesses the cell, returning a pointer that the client may freely use. In, free the code first checks that its argument is in range. If it is, it checks that the cell has previously been allocated by the allocator and not yet freed (meta[i].tag == 1). Next, it checks that the data is not still (erroneously) in use by another module at a protected type by testing if data[i] is vacant (line 23). Finally, if all these checks succeed, the metadata is set to unallocated and the data is blessed, protecting it from use by any other module.

When thinking about the correctness of alloc and free, the first thing to notice is that if the informal invariants mentioned above are true at entry to either routine then they are also true upon completion of the routine. More interesting still, the invariants (though loosely stated) are phrased entirely in terms of protected state — *i.e.,* in terms of static global arrays, whose addresses may not be changed, in terms of protected memory, such as the contents of meta, and in terms of a locally quantified variable i — as opposed to in terms of normal, vulnerable, heap-allocated data structures. Because these invariants depend exclusively on protected state, no client module may corrupt them and hence, according to the traditional *hypothetical frame rule*, if initialization (not shown) makes them valid at the outset, it is sound for each routine to depend upon their continued validity throughout the program.

3. Semantics of YARRA

This section defines YCORE, a sequential, imperative language intended to serve as a core model for YARRA. Our main technical contribution is a logic for YCORE programs that supports local reasoning via a frame rule, even in the presence of unverified third-party code that can have arbitrary effects on the heap. We prove our logic sound against a dynamic semantics for YCORE that maps directly to the library protection mode of our implementation.

Modeling critical types using partial maps and backing stores. Abstractly, the YCORE run-time system consists of a set of independent, isolated heaplets, one for each critical type X, and the conventional heap (H). The heaplet for X corresponds roughly to the backing store for X-typed objects. We model the critical heaplets formally as partial maps from memory addresses to X-typed objects. The conventional heap H is a total map from memory addresses to integers. Declaring a new type X corresponds to allocating a new map, also named X (*i.e.*, reserving space in the backing store for X-typed objects); blessing an object as X corresponds to adding a value to the X map (*i.e.*, making a copy of it in the backing store); unblessing removes an element from the map (*i.e.*, reclaiming space in the backing store). Checked reads and writes access the fields of an object by consulting these maps, while untrusted reads and writes performed by library code merely access the untyped heap, without any impact on the maps for object types. While it is convenient to think of critical heaplets in terms of backing stores, these stores need not actually be materialized at runtime. Indeed, the source protection mode of our compiler implements the semantics of backing stores indirectly via code instrumentation.

3.1 Syntax

Figure 2 shows the syntax of YCORE, starting with our meta variable conventions. Values in YCORE are either integer constants i, or are structured tuples (v_1, v_2) corresponding to the values of protected object types. Expressions e are purely arithmetic, and the statement language s includes the usual commands for branching, looping, sequencing, and assertions. The assertion logic of YCORE makes use of first-order formulas Φ over a term language including arithmetic expressions, tuples, maps and sets, together with (extensional) equality, set membership, and integer inequality.

In addition to the basic commands, YCORE includes scoped, local variable declarations, (local x in s)—local variables always hold integer values, so no type is needed on the declaration of x. The form (newtype $X = \tau$ in s) allows us to define a name X for a new critical type, where the representation of X is τ, and X can be used in s. The statements for blessing and unblessing are slightly more general than what was used in Section 2. Here, bless and unbless operate on a range of locations starting at e_{base} and including e objects each to either be protected or unprotected at the type X (where e is expected to evaluate to a non-negative integer). The dynamic typecase statement is useful for modeling the vacant command of Section 2.2, as well as other constructs—it can be used

integer constants			i, j, ℓ
local variables			x, y, z
map type names			X, Y, Z, H, Un
values	v	::=	$i \mid (v_1, v_2)$
expr.	e	::=	$i \mid x \mid e\ op\ e'$
stmt.	s	::=	skip $\mid$ if e then s_1 else $s_2 \mid$ while $e\ s \mid s_1; s_2$
assertion			$\mid$ assert Φ
local var. decl.			$\mid$ local x in s
local type decl.			$\mid$ newtype $X = \tau$ in s
bless e objs. starting at e_{base}			$\mid y :=$ bless$_X [e]\ e_{\text{base}}$
unbless e objs. starting at e_{base}			$\mid y :=$ unbless$_X\ [e]\ e_{\text{base}}$
dynamic typecase			$\mid$ if e is in X then s_1 else s_2
checked read			$\mid y := X(e).p$
unchecked read			$\mid$ lib $y := e$
checked write			$\mid X(e_1).p := e_2$
unchecked write			$\mid$ lib $e_1 := e_2$
dynamic failure			$\mid$ abort
field path	p	::=	$\cdot \mid 0p \mid 1p$
types	τ	::=	$int \mid (\tau_1, \tau_2) \mid X$
map type	$\hat{\tau}$	::=	$int \to \tau$
map value	$\hat{v}$	::=	$\lambda\ell.\hat{e}$
map body	$\hat{e}$	::=	$\bot \mid v \mid \hat{v}\ v \mid$ if $a \in a'$ then $\hat{e}$ else $\hat{e}'$
mod. set	Δ	::=	$\cdot \mid \Delta, X \mid \Delta, x$
static env.	Γ	::=	$\cdot \mid \Gamma, X{:}\hat{\tau} \mid \Gamma, x$
logic term	a	::=	$e \mid v \mid \hat{e} \mid \hat{v} \mid X \mid a.p \mid dom\ a \mid \{x \mid \Phi\}$
formula	Φ, Ψ	::=	$\Phi \wedge \Psi \mid \Phi \vee \Psi \mid \neg\Phi \mid \forall x.\Phi \mid \forall X{:}\hat{\tau}.\Phi$
			$\mid a = a' \mid a \in a' \mid a < a' \mid True \mid False$
substitution	σ	::=	$\cdot \mid \sigma, [a/X] \mid \sigma, [a/x]$

Figure 2. Syntax of YCORE

to check whether a location e holds a critical object of type X. We include two forms of read and write instructions. A checked read attempts to read a structured value v of type X at the location e and projects a field from v using the path p, storing the result in the local variable y. In contrast, an unchecked read instruction reads the contents of an arbitrary memory location from the heap into a local variable. Similarly, a checked write attempts to write to a structured type using a field assignment; unchecked writes modify a single location in the heap. We use the unchecked forms to model the actions of arbitrary, untrusted code, *e.g.*, third party libraries. Such libraries are assumed to contain an arbitrary sequence of unchecked reads and writes, in combination with branching, looping *etc.*, but, importantly, never contain bless, unbless, typecase or checked read and write commands.

We model two kinds of failure modes in YCORE. Certain dynamic failures are permitted by the logic, *e.g.*, failures caused by the effects of untrusted libraries which are detected by the runtime system. These failures cause a program to loop indefinitely issuing the abort command—we expressly choose to allow such "safe" failures to occur at run time since they are unavoidably triggered by the behavior of unverified library code. Other failures, *e.g.*, trying to bless a piece of memory that has already been blessed at another type, or an assertion failure, cause the program to get stuck. YCORE's logic is designed to prevent the latter form of failure.

The type language of YCORE includes int, pairs, and critical type names X. We model both C's integers as well as pointers using the int type. Structures in C, which contain an arbitrary number of named fields, are modeled using pairs. We omit unions.

Type names X in YCORE also double as the names of map values $\lambda\ell.\hat{e}$, lambda-terms that map integer locations to values and that model the backing store for X objects. The body of a map expression is built from values v, an application form, a conditional form, and $\bot$. We use the latter to model partial maps.

```
1 yarra struct {int f0; int f1} X;
2 yarra struct {X g0; int g1} Y;
3 main() {
4   void* z=malloc(sizeof(Y));
5   X* x = bless<X>(1, z);
6   Y* y = bless<Y>(1, z);
7   y.g0.f0 = 17;
8   void * _ = unbless<Y>(1, y);
9   void * _ = unbless<X>(1, x); }
```

```
newtype X = (int, int) in
newtype Y = (X, int) in
local x, y, z in
  z := ℓ;
  x := bless_X [1] z;
  y := bless_Y [1] z;
  Y(y).00 := 17;
  _ := unbless_Y [1] y;
  _ := unbless_X [1] x
```

Figure 3. Relating the syntax of YARRA to YCORE

$X[a \leftarrow a']$	=	$\lambda\ell$.if $\ell \in \{a\}$ then a' else $(X\ \ell)$
$range_\Gamma\ X$	=	τ **when** $\Gamma(X) = int \to \tau$
$\lvert int\rvert_\Gamma$	=	1
$\lvert Y\rvert_\Gamma$	=	$\lvert range_\mathcal{E}\ Y\rvert_\Gamma$
$\lvert(\tau_1, \tau_2)\rvert_\Gamma$	=	$\lvert\tau_1\rvert_\Gamma + \lvert\tau_2\rvert_\Gamma$
$offset_\Gamma\ int\ \cdot$	=	0
$offset_\Gamma\ (\tau_1, \tau_2)\ 0p$	=	$offset_\Gamma\ \tau_1\ p$
$offset_\Gamma\ (\tau_1, \tau_2)\ 1p$	=	$\lvert\tau_1\rvert_\Gamma + offset_\Gamma\ \tau_2\ p$
$offset_\Gamma\ Y\ p$	=	$offset_\Gamma\ (range_\Gamma\ Y)\ p$
$readFrom_\Gamma\ Y\ (\ell{:}int)$	=	$(Y\ \ell)$
$readFrom_\Gamma\ Y\ (\ell{:}Z)$	=	$readFrom_\Gamma\ Y\ (\ell{:}(range_\Gamma\ Z))$
$readFrom_\Gamma\ Y\ (\ell{:}(\tau_1, \tau_2))$	=	(v_1, v_2)
where $v_1 = readFrom_\Gamma\ Y\ (\ell{:}\tau_1)$		
and $v_2 = readFrom_\Gamma\ Y\ ((\ell + \lvert\tau_1\rvert_\Gamma){:}\tau_2)$		

Figure 4. Auxiliary functions

Figure 3 illustrates how the concrete syntax of YARRA maps to YCORE. Struct declarations correspond to declarations of tuple types. We do not include procedures in YCORE—the statement s can be thought of as the body of `main`. We also do not provide primitive operations for dynamic memory allocation in YCORE—so the `malloc` call at line 4 has no direct analog in YCORE. However, we model the heap as a total map over integer locations and we can *program* `malloc` in YCORE.[3] In this example, which will be reused later to illustrate the static semantics, we replace the call to `malloc` with an abstract address ℓ. Calls to `bless` and `unbless` in YARRA map directly to YCORE. Writes and field projections via object references in YARRA also map directly, as shown on line 7. YCORE uses binary paths to the fields of tuples, instead of field names. More importantly, while writes to objects via typed references in YARRA are evident from the declared types (for example, the type `Y*` of `y`), in YCORE, the write instruction itself is tagged with the type of the object that is the destination of the write. Typed read instructions are similar. For convenience, our example hoists the local variable declarations.

3.2 Static semantics

The static semantics of YCORE is presented as a classical Floyd-Hoare logic. Before explaining the details of the logic, however, we must comment on several bits of notation. First, the judgement depends upon a context Γ, that contains the mapping of type names X to their map types $\hat{\tau}$ and the set of local variables x that are in scope. Well-formedness conditions on Γ ensure that it always contains bindings for two distinguished map variables: H a total map from integer locations to integer values, which represents the conventional heap; and Un, a partial map whose domain is precisely the set of unprotected locations. Second, the semantics depends upon several auxiliary functions shown in Figure 4. Most of these functions are straightforward, although two comments

[3] This is not an unusual choice in systems governed by classical logics. See, for example, work on Havoc [13].

$$\boxed{\Gamma;\Delta \vdash \{\Phi\}\ s\ \{\Psi\}} \qquad \frac{\Gamma;\Delta \setminus FV(\Phi') \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma;\Delta \vdash \{\Phi' \wedge \Phi\}\ s\ \{\Phi' \wedge \Psi\}}\ \text{T-Frame} \qquad \frac{\Gamma \vdash \tau\ \text{ok} \quad X \notin dom\ \Gamma \quad \hat{\tau} = int \to \tau \quad \Gamma, X{:}\hat{\tau}; \Delta, X \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma;\Delta \vdash \{\forall X{:}\hat{\tau}.X = \lambda\ell.\bot \Rightarrow \Phi\}\ \text{newtype}\ X = \tau\ \text{in}\ s\ \{\Psi\}}\ \text{T-NewX}$$

$$\frac{\Gamma \vdash e_1, e_2, y\ \text{ok} \quad L = \bigcup_{0 \le i < e_1}\{e_2 + |X|_\Gamma * i\} \quad range_\Gamma\ X = \tau \quad y, X, Un, \tau \in \Delta \quad \sigma_1 = copy_\Gamma\ L\ from\ H\ to\ X \quad \Phi, \sigma_2 = chkAndRem_\Gamma\ \tau\ L \quad \sigma_3 = updUn_\Gamma\ L\ \tau\ \bot}{\Gamma;\Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := \text{bless}_X[e_1]\ e_2\ \{\Psi\}}\ \text{T-Bless} \qquad \frac{\Gamma \vdash \Phi\ \text{ok}}{\Gamma;\Delta \vdash \{\Phi \wedge \Psi\}\ \text{assert}\ \Phi\ \{\Psi\}}\ \text{T-Assert}$$

$$\frac{\Gamma \vdash e_1, e_2, y\ \text{ok} \quad L = \bigcup_{0 \le i < e_1}\{e_2 + |X|_\Gamma * i\} \quad range_\Gamma\ (X) = \tau \quad y, X, Un, \tau \in \Delta \quad \sigma_1 = copy_\Gamma\ L\ from\ H\ to\ \tau \quad \Phi, \sigma_2 = chkAndRem_\Gamma\ X\ L \quad \sigma_3 = updUn_\Gamma\ L\ \tau\ 1}{\Gamma;\Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := \text{unbless}_X\ [e_1]\ e_2\ \{\Psi\}}\ \text{T-UnBless} \qquad \frac{}{\Gamma;\Delta \vdash \{True\}\ \text{abort}\ \{\Psi\}}\ \text{T-Abort}$$

$$\frac{\Gamma \vdash e\ \text{ok} \quad v_h = readFrom_\Gamma\ H\ (e{:}X) \quad v_x = X_\Gamma(e) \quad \Gamma;\Delta \vdash \{\Phi_1\}\ s_1\ \{\Psi\} \quad \Gamma;\Delta \vdash \{\Phi_2\}\ s_2\ \{\Psi\}}{\Gamma;\Delta \vdash \{((e \in dom_\Gamma\ X \wedge (X = Un \vee v_h = v_x)) \Rightarrow \Phi_1) \wedge (e \notin dom_\Gamma\ X \Rightarrow \Phi_2)\}\ \text{if}\ e\ \text{is in}\ X\ \text{then}\ s_1\ \text{else}\ s_2\ \{\Psi\}}\ \text{T-IsX}$$

$$\frac{\Gamma \vdash e, y\ \text{ok} \quad y \in \Delta \quad X \neq Un \quad v_h = readFrom_\Gamma\ H\ (e{:}X) \quad v_x = X_\Gamma(e) \quad \sigma = [(H_1(e + offset_\Gamma\ X\ p))/y]}{\Gamma;\Delta \vdash \{e \in dom_\Gamma\ X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ y := X(e).p\ \{\Psi\}}\ \text{T-Rd} \qquad \frac{\Gamma \vdash e, y\ \text{ok} \quad \sigma = [(H\ e)/y]}{\Gamma; y \vdash \{\sigma(\Psi)\}\ \text{lib}\ y := e\ \{\Psi\}}\ \text{T-LibRd}$$

$$\frac{\Gamma \vdash e_1, e_2\ \text{ok} \quad X, H \in \Delta \quad X \neq Un \quad v_h = readFrom_\Gamma\ H\ (e_1{:}X) \quad v_x = X_\Gamma(e_1) \quad H_1 = H[(e_1 + offset_\Gamma\ X\ p) \leftarrow e_2] \quad \sigma_1 = copy_\Gamma\ e_1\ from\ H_1\ to\ X \quad \sigma = \sigma_1 \circ [H_1/H]}{\Gamma;\Delta \vdash \{e_1 \in dom_\Gamma\ X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ X(e_1).p := e_2\ \{\Psi\}}\ \text{T-Wr} \qquad \frac{\Gamma \vdash e_1, e_2\ \text{ok} \quad H_1 = H[e_1 \leftarrow e_2] \quad \sigma = [H_1/H]}{\Gamma; H \vdash \{\sigma(\Psi)\}\ \text{lib}\ e_1 := e_2\ \{\Psi\}}\ \text{T-LibWr}$$

copy-from-to : $(Env * Locs * Map * Type) \to Subst$

$$\begin{array}{lcl} copy_\Gamma\ L\ from\ Y\ to\ int & = & \cdot \\ copy_\Gamma\ L\ from\ Y\ to\ X & = & [(\lambda\ell.\text{if}\ \ell \in L\ \text{then}\ (readFrom_\Gamma\ Y\ (\ell{:}X))\ \text{else}\ X\ \ell)/X] \\ copy_\Gamma\ L\ from\ Y\ to\ (\tau_1, \tau_2) & = & \text{let}\ \sigma_1 = copy_\Gamma\ L\ from\ Y\ to\ \tau_1\ \text{in} \\ & & \text{let}\ \sigma_2 = copy_\Gamma\ \{\ell + |\tau_1|_\Gamma \mid \ell \in L\}\ from\ Y\ to\ \tau_2\ \text{in} \\ & & \sigma_1 \circ \sigma_2 \end{array}$$

Membership of types in the modifies set, Δ

$$\begin{array}{lcl} int \in \Delta & = & True \\ X \in \Delta & = & \exists \Delta_1, \Delta_2.\Delta = \Delta_1, X, \Delta_2 \\ (\tau_1, \tau_2) \in \Delta & = & \tau_1 \in \Delta \wedge \tau_2 \in \Delta \end{array}$$

chkAndRem : $(Env * Type * Locs) \to (Prop * Subst)$

$$\begin{array}{lcl} chkAndRem_\Gamma\ int\ L & = & (L \subseteq dom\ Un, \cdot) \\ chkAndRem_\Gamma\ X\ L & = & \text{let}\ \Phi = \forall x.x \in L \Rightarrow x \in dom_\Gamma\ (X)\ \text{in} \\ & & (\Phi, [(\lambda\ell.\text{if}\ \ell \in L\ \text{then}\ \bot\ \text{else}\ X\ \ell)/X]) \\ chkAndRem_\Gamma\ (\tau_1, \tau_2)\ L & = & \text{let}\ \Phi_1, \sigma_1 = chkAndRem_\Gamma\ \tau_1\ L\ \text{in} \\ & & \text{let}\ \Phi_2, \sigma_2 = chkAndRem_\Gamma\ \tau_1\ \{\ell + |\tau_1|_\Gamma \mid \ell \in L\}\ \text{in} \\ & & (\Phi_1 \wedge \Phi_2, \sigma_1 \circ \sigma_2) \end{array}$$

updUn : $(Env * Locs * Type * MapBody) \to Subst$

$$\begin{array}{lcl} updUn_\Gamma\ L\ int\ \hat{e} & = & [\lambda\ell.\text{if}\ \ell \in L\ \text{then}\ \hat{e}\ \text{else}\ Un\ \ell/Un] \\ updUn_\Gamma\ L\ X\ \hat{e} & = & \cdot \\ updUn_\Gamma\ L\ (\tau_1, \tau_2)\ \hat{e} & = & \text{let}\ \sigma_1 = updUn_\Gamma\ L\ \tau_1\ \hat{e}\ \text{in} \\ & & \text{let}\ L_1 = \{\ell + |\tau_1|_\Gamma \mid \ell \in L\}\ \text{in} \\ & & updUn_\Gamma\ L_1\ \tau_2\ \hat{e} \end{array}$$

Figure 5. A Floyd-Hoare logic for YCORE (Omitting rules for standard constructs)

are worthwhile. First, note that $offset_\Gamma\ \tau\ p$ is a partial function, *e.g.*, $offset_\Gamma\ ((int, int), int)\ 0$ is undefined. This ensures that only word-length int-valued fields in a nested tuple type can be directly addressed. Second, $readFrom_\Gamma\ Y\ (\ell{:}\tau)$ is used to read a structured value of type τ from the location ℓ in the map Y. While this function is well-defined for arbitrary maps Y, we use it primarily to read structured values out of the flat heap map H.

With these definitions in hand, the reader may turn to Figure 5, which presents the main semantic rules for YCORE. For space reasons, this figure omits several rules including rules for branching, loops, sequencing, skip, local variables, and the rule of consequence. Our auxiliary submission materials include these omissions. The central judgment presented in the Figure, $\Gamma;\Delta \vdash \{\Phi\}\ s\ \{\Psi\}$ states, informally, that when executed in an environment E modeled by the context Γ, and when E satisfies the pre-condition Φ, the program s, if it terminates, produces some environment E' that satisfies the post-condition Ψ, while modifying at most the variables in the set Δ. The following paragraphs explain the key rules.

The frame rule. The key feature of our logic is that it admits the frame rule, (T-Frame), which states that a formula Φ', whose free variables do not overlap with the set of free variables modified by a statement s, is preserved across execution of s. Crucially, *because the state of critical data with type X is represented with a variable X that is distinct from variable H, the frame rule can soundly be used to preserve invariants of that critical data, when X is unmodified, despite arbitrary modifications to H in s.*

Declaring new types. (T-NewX) shows how new types are introduced. The premises of the rule check that the type τ is well-formed (*e.g.*, does not mention names that are not in scope) and that X is a fresh name. The body s is checked in a context where X is bound to the type of a map, and X is recorded as one of the variables that may be modified by s. Since all new type maps are initially empty, the pre-condition of s may be proven under the assumption that $X = \lambda\ell.\bot$.

Blessing and unblessing. The rules (T-Bless) and (T-UnBless) are closely related—in fact, they are symmetric. The command $y := \text{bless}_X[e_1]\ e_2$ blesses a sequence of e_1 objects beginning at e_2 to the type X, *i.e.*, it casts e_2 to the base of an e_1-numbered array of X objects and stores a reference to the base location in the local variable y. The unbless command does the opposite, removing the protection on an array of objects. We illustrate the behavior of these operations using the YCORE program in Figure 3.

This program declares two object types X and Y, where the type Y has the type X nested within its first component. When blessing an object Y, YARRA requires all sub-objects of Y to already be blessed. Since every Y object contains an X object as a prefix, the we must bless the contained X object first. The program above does just this, by first blessing the memory location ℓ as containing a single X object, and then blesses the location ℓ again as a Y object.

Abstractly, we model this behavior by allocating two maps corresponding to the types X and Y. At the first bless command, (T-Bless) computes the set L of locations in the array to be blessed.

In our example, this is just the singleton set $\{\ell\}$. Using the function *copy*$_\Gamma$ L *from* H *to* X, we read X-typed tuple values from the heap H at each location in L into the backing store for X, the map X. At the first bless command in our example, this corresponds to reading $v_x = (H\ \ell, H\ (\ell+1))$ and adding it to the X map at location ℓ. At the second bless command, we copy the value $v_y = (v_x, H\ (\ell+2))$ (a Y-typed value) into the map Y at location ℓ.

Additionally, when blessing locations we enforce two other invariants key to the soundness of our frame rule. First, when blessing a location ℓ to be a type τ, we must check that the fields of the type τ are appropriately blessed or unblessed—we call this the *field consistency* condition. For this purpose, in addition to the maps for each type, our semantics also keeps track of a map $Un : int \to int$ for locations that are not blessed at any protected type. Second, we ensure that in addition to the heap H, every memory location is in at most one map—we call this the *disjoint domains* condition.

We use two auxiliary functions to enforce these invariants. At the first bless command of our example, *chkAndRem*$_\Gamma$ (int, int) $\{\ell\}$ checks that the locations $\{\ell, (\ell+1)\}$ are currently unblessed, *i.e.*, they are in the *Un* map. At the second bless command, we use *chkAndRem*$_\Gamma$ (X, int) $\{\ell\}$ to check that location ℓ is in the domain of X and location $(\ell+2)$ is unblessed. In both cases, the check manifests itself as a pre-condition Φ for verifying the bless command. For the second bless, to ensure the maps for X and Y do not overlap, we additionally compute a substitution σ_2 which updates the map X by removing the location ℓ from its domain. The function *updUn*$_\Gamma$ $L\ \tau\ \perp$ computes a substitutions that removes locations that are newly blessed from the *Un* map—at the first bless these locations are $\{\ell, \ell+1\}$ and, at the second, $\{\ell+2\}$.

Finally, we require y, X and *Un* to be in the set of modified locations Δ. Additionally, since the maps of nested types are also modified (*e.g.*, the map X when blessing a location as Y), we overload notation and require τ to also be in Δ. The pre-condition in the conclusion is a propagation of the post-condition under the composition of all the computed substitutions. We also include the formula Φ in the pre-condition to enforce field consistency.

The rules for unbless are entirely symmetric to those for bless, swapping the role of a type name X for its representation τ, and adding elements to the *Un* map instead of removing them. In our example, the first unbless removes a value $v_y = (v'_x, i)$ from the Y-map at location ℓ; adds v_x to X at location ℓ, and adds the location $\ell+2$ back to the *Un* map. The second unbless removes v'_x from X at location ℓ and adds $\{\ell, \ell+1\}$ back to the *Un* map.

Typecase. The typecase construct allows a programmer to test whether a location is either the head of an X-typed object, or not blessed at all. To test the latter condition, a programmer can write (if e is in *Un* then s_1 else s_2), which causes s_1 to be executed only if e is an unblessed location—this is a primitive form of the `vacant` function used in the memory manager of Section 2.2, which can be expanded to a sequence of typecase commands. (T-IsX) formalizes the semantics of typecase. The then-branch s_1 can assume that the scrutinee e is in the backing store of X and, when X is not *Un*, can additionally assume that the value of X in the backing store matches the contents of the heap H. A mismatch between the backing store and heap signals a potential corruption of memory by library code—this situation is detected dynamically by the YARRA runtime and causes the program to abort. The else-branch, in contrast, can assume that e is not in X.

Checked reads. A checked read is modeled using the instruction $y := X(e).p$, where X can be any type name in scope or the heap H, but not the *Un* map. Although (T-Rd) is uniform with regard to the choice of X, it is instructive to first examine the case where X is not H. In this case, the pre-condition includes a check to ensure that e is in the domain of X, in effect checking that e is indeed a reference to an object of type X, as claimed by the program text. As in the (T-IsX) rule, the semantics allows us to assume that the value of e in the backing store v_x, and its value in the heap v_h are synchronized—a mismatch results in an abort. Note that although we are reading a single integer-valued field from the object, (T-Rd) allows us to assume that the entire object v_h is synchronized with v_x, *i.e.*, protections in YARRA operate at a level of granularity corresponding to the object, allowing programmers to reason about and preserve internal invariants among the fields of an object, rather than each field in isolation. A checked read from the heap (when $X = H$) is less interesting—both conditions in the conclusion that guard the implication are tautologies. In other words, YARRA provides no guarantees for programs that read from the heap using untyped pointers.

Un-checked reads. The command lib $y := e$ is a read instruction performed in unverified third-party code. We provide no special semantics for this command—libraries are free to read from arbitrary portions of the heap.

Checked writes. The rule (T-Wr) is analogous to (T-Rd). It requires the programmer to show that the location e_1 being written to is indeed a reference to an X-typed object. In return, it allows the programmer to assume that the value in the heap v_h matches the value in the backing store v_x. Additionally, we propagate the post-condition Ψ to account for an update to the heap H, as well as the backing store X.

Un-checked writes. The command lib $y := e$ is a write instruction performed in unverified third-party code. (T-LibWr) shows that it is free to modify arbitrary heap locations, but never a location in one of the X or *Un* maps. In effect, the rule says that writes by libraries can corrupt the heap, but that the integrity of the backing stores are always preserved, thereby allowing the runtime system (however it chooses to implement the backing store) to detect (and possibly recover from) memory corruptions due to libraries.

Libraries and the frame rule. As mentioned earlier, library code contains unchecked reads and writes and arbitrary control-flow but not the special YARRA commands. A careful inspection of the static semantics reveals that these unchecked reads and writes construct preconditions from postconditions by performing a substitution as opposed to demanding validity of a logical condition. Hence, any well-scoped combination of such commands s satisfies the *trivial Hoare triple* $\{True\}s\{True\}$ and modifies no type maps X aside from H. Hence, suppose Φ contains only references to types X and local variables x inaccessible to the library s. In such a case, according to the frame rule, Φ is preserved across calls to s. Most importantly, we can come to the conclusion that Φ is preserved *without having to analyze or modify the memory access patterns of* s. Therein lies the power of YARRA.

3.3 Soundness of the logic

We prove our logic sound with respect to a dynamic semantics that maintains explicit backing stores for each object type. Our dynamic semantics is a relation of the form $(E; s) \rightsquigarrow (E'; s')$, representing a small step of reduction of a runtime configuration $(E; s)$ consisting of a statement s, and a runtime environment E, which contains map values for the heap H, the *Un* map, and for each of several types X and local variables x that are in scope for s. Our soundness results come in the form of progress and preservation results that guarantee that YCORE programs that verify according the logic never get stuck (although they may abort).

Theorem 1 (Soundness)**.** *For all environments* Γ, Δ *(such that* $\vdash \Gamma; \Delta$ *ok); formulas* Φ, Ψ *(such that* $\Gamma \vdash \Psi$ *ok); stores* E *(such that* $\vdash E : \Gamma$ *and* $E \models \Phi$*); and programs* s *such that* $\Gamma; \Delta \vdash \{\Phi\}\ s\ \{\Psi\}$*. If* $s \neq$ *skip, then there exists* $E', s', \Gamma'', \Phi', \Delta'$

such that 1) $(E, s) \rightsquigarrow (E'; s')$*; 2)* $\vdash E' : \Gamma, \Gamma''$*; 3)* $\Gamma, \Gamma''; \Delta' \vdash \{\Phi'\}\ s'\ \{\Psi\}$*; and 4)* $E' \models \Phi'$.

4. Implementation

The YARRA compiler is implemented as a plug-in to the CIL compiler infrastructure [17]. It implements YARRA's protection mechanisms using two sets of techniques. YARRA *source protections* rewrite C source code under compiler control to ensure that the program does not incorrectly access critical data types. YARRA *library protections* use a backing store to ensure that libraries, whose source we cannot rewrite, will be unable to corrupt critical data.

4.1 YARRA Source Protections

YARRA source protections rely on compiling the entire program with the YARRA compiler. At runtime, each memory location is assigned a YARRA *type identifier* (a ytype) corresponding to the type of data it holds. The bless and unbless instructions change the ytype associated with a set of locations. Read and write instructions are instrumented with checks to ensure that the static types of the pointers involved match the ytype associated with the memory locations read from or written to.

The runtime system maintains the type information and implements the checks. The key data structure is a map that associates each memory address a with a pair consisting of a bit and a ytype. The bit marks whether a is the head of some critical object, and the ytype identifies the type of the enclosing critical object. If the location is not part of an object, its ytype is Un. The runtime system exposes the following functions that manipulate the map.

Bless: void bless<ytype t>(void *p). The bless function sets the head bit at location p and assigns t to each location in [p, p + sizeof(t)). It requires fields of p with critical types to be blessed in advance; the type identifiers of the nested objects are replaced by t and their head bits are reset.

Typecase: int isIn<ytype t>(void *p). Typecase is implemented as a boolean function, which returns a non-zero integer if p has been blessed with type t—*i.e.,* the head bit is set and locations [p, p + sizeof(t)) have ytype t.

Unbless: void unbless<ytype t>(void *p). The unbless function undoes the effects of bless. First, it calls isIn(t, p) to ensure that p has been previously blessed. Second, it clears the head bit and ytype from [p, p + sizeof(t)) and restores the head bits and ytype for any fields with critical types.

Vacant: int vacant<ytype t>(void *p). The vacant function returns a non-zero integer if [p, p + sizeof(t)) has ytype Un.

The YARRA compiler does the following:

- Builds run-time type representations for each critical type. Each representation includes the ytype, its size, and offsets of fields.
- Prefaces each critical read and write of pointer p with a call to isIn(typeOf(p), p). Execution aborts if the call fails.
- Prefaces each untyped write with a call to vacant and aborts if it returns 0.

4.2 YARRA Library Protections

YARRA Library protections rely on (1) maintaining a *backing store* that stores copies of critical data, and (2) protecting that backing store from library access.

Maintaining the Backing Store. The backing store is realized by adding a third field to the map described in Section 4.1. In other words, when library protections are enabled, the range of the map is a triple of a bit, a ytype, and a shadow byte. The shadow byte stores a copy of the value at the address in question. The runtime functions are similar to those in Section 4.1, with the following changes.

- ***Typecase.*** The implementation of isIn is augmented to compare the value of shadow with the value at address in the heap. If the address has been blessed and the comparison detects a difference, indicating a potential corruption, isIn aborts the program. Notice that since the implementation of critical reads and writes use isIn, they only succeed when the shadow copy is in synch with the ordinary copy.
- ***Bless.*** bless is augmented to copy values of newly blessed addresses to the backing store.

Also, critically-typed writes are instrumented at runtime with a call to a new runtime function, yShadowWrite(void *p, size_t size), which copies the values in the heap starting at p into the backing store.

Protecting the Backing Store. The backing store uses a special critical memory manager, implemented using the BGET memory manager [25], for memory allocations. The memory pool given to BGET is tracked, and the YARRA runtime system exposes yUnlock(void) and yLock(void) functions for setting and unsetting write permissions on those pages respectively. Boundary crossings from protected to unprotected functions are instrumented with calls to yLock(), and each function in the runtime API calls yUnlock if the backing store has been locked, effectively unlocking on demand.

Implementation Options. We implement two versions of the address map: a standard hash table and a two-level lookup table—the latter is similar to that used by Valgrind [19]. Although the space overhead of both implementations grows linearly with the number of blessed locations, the space overhead of the hash table is much smaller. However, the number of reads required by each hash table lookup is proportional to the number of hash collisions, and hence its efficiency degrades as the number of blessed locations increase. The hash table implementation is thus well suited for YARRA-protected programs with few blessed locations and many boundary crossings.

The lookup table uses a primary table with 64K entries, each of which points to a secondary table with 64K tuples. This associates a tuple with each byte in memory; the higher order 16 bits determine the offset in the primary table, and the lower order bits identify a tuple in the secondary table. Secondary tables are only allocated when a byte within their range is blessed, making unblessed lookups very fast and blessed lookups slightly slower. However, the primary and all secondary tables must be protected on every boundary crossing, which can be expensive, given that the primary table alone requires 2^{18} bytes. Thus, the two-level page table implementation is well suited to programs with many blessed locations and fewer boundary crossings.

5. Evaluation

In this section, we evaluate our prototype implementation of YARRA. The important take-away is that despite our naive implementation, YARRA's performance is already entirely adequate to protect small sets of high-value data structures, and that in doing so, YARRA can defend against important vulnerabilities with negligible impact on end-to-end application performance. Alternative approaches based on array-bounds checking cannot (soundly) implement such targeted, negligible performance protections.

5.1 YARRA Applications

We consider two different use cases for YARRA and evaluate each, using the programs in Figure 6. The first involves having YARRA protect module data structures so that clients may not corrupt it. We study this use case primarily through experimentation with the BGET memory allocator [25]. We use YARRA to protect BGET's metadata from clients that use the allocator in a way reminiscent of the idealized allocator example presented in Section 2.2. The

Program	YARRA Protections	Orig. LOC / Mod. LOC	Bless / Unbless
BGET	Allocator metadata.	241 / 43	16
sshd	Password structure and validation bit.	60148 / 497	23
ftpd	Path/command buffers.	17993 / 262	3
ghttpd	Pointer to command buffer.	514 / 69	3
telnetd	Login command string.	3962 / 63	3

Figure 6. YARRA-protected Applications

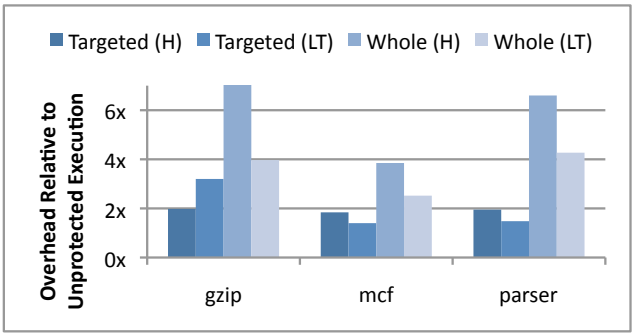

Figure 7. CPU overhead for securing allocator metadata using YARRA's targeted and whole program protection modes with both hash table (H) and look-up table (LT) implementations of the address map. A value of *1x* indicates no measurable overhead.

BGET clients we measure are three SPECINT2000 programs also used in the WIT paper [2]. (The SPEC benchmarks are not included in the figure because they were not modified, only linked against a modified version of BGET.) Because these clients frequently call allocation and deallocation routines which contain bless and unbless operations, this case study exercises our implementation vigorously. This experiment is also interesting because it shows that YARRA can simulate the protections provided by recent work [12] that uses a separate process to protect heap metadata

The second use case investigates YARRA's potential for securing specific, known data vulnerabilities in server applications. The programs we use, daemons for several common network protocols, are taken from previous work documenting non-control data attacks [5] and are shown in Figure 6. In each case, YARRA secures a small number of high-value data items, which are known to be vulnerable to attacks. Figure 6 shows the nature of data protected in each application, as well as the extent of the modifications required to protect it, and the number of bless and unbless instructions inserted.

As the table indicates, it was not difficult to introduce YARRA protections in these applications. Few locations required blessing and unblessing, and the vast majority of modified lines were changed by automated search and replace of variable names. Each application required less than a day's effort to protect with YARRA.

5.2 Performance Overhead

Protecting internal state. We can run YARRA in two modes. In *whole program protection* mode, we use the source protections defined in section 4 on the entire application. In *targeted protection* mode, we use source protections on the core routines and treat the rest of the application as a library, incurring a boundary crossing cost, but otherwise leaving the library untouched.

Figure 7 compares the overhead of whole program protection to that of targeted protection with the hash table and lookup table versions of the address map.[4] We measure overhead relative to unprotected execution; a value of *1x* indicates no measurable difference. Targeted protection is more efficient with these applications, indicating that the cost of boundary crossings is less than instrumenting every read and write in the application.

The address map implementation has a clear impact on whole program protection. Because whole program protection relies entirely on read and write instrumentation, the size of the address map is much less important than look-up speed, and hence the LT implementation is faster. However, the difference is less pronounced with targeted protection. Although look-up speed is still important, a larger address map increases the cost of boundary crossings, because the table needs to be protected.

Even with targeted protection, the overhead ranges from 50% to 200%. Protecting allocator metadata is an interesting challenge, because ownership of memory is interleaved at a fine granularity and changes dynamically. We found that the number of reads and writes instrumented, calls to bless/unbless, and boundary crossings all scaled linearly with the number of objects allocated (not shown).

Protecting vulnerable data. We applied YARRA targeted protections to known data vulnerabilities in four server applications and evaluated the performance impact. The interactive nature of the applications prompted us to measure the performance impact on execution within the protected module, from the server's perspective, as well as from the client's perspective.

Figure 8 shows the performance overhead within the hardened module measured relative to unprotected execution. [5] This overhead includes both the cost of initializing the YARRA run-time system and that of boundary crossings. A value of *1x* indicates no measurable overhead. We found that the number of reads and writes per boundary crossing was relatively low for these applications, which highlights the performance impact of the address map implementation – the small size of the hash table more than made up for the slower table accesses.

Using the hash table address map, we saw the performance overhead in the protected module range from 7% to 67%. The protected module of telnetd included setting up a socket connection, which dominated our measurement, and hence the measured overhead of telnetd was negligible.

Figure 8 also shows the performance overhead from the client's perspective. We found no measurable difference between connecting to a vulnerable server and a hardened server. The clients and servers were run on the same machine to minimize connection latency, and each client performed a routine task that touched formerly vulnerable data.

Optimizations. Read/write instrumentations and boundary crossings are both bottlenecks in our current implementation. Because our implementation is not as highly optimized as other, similar bounds-checking implementations (e.g. [16, 21]), we anticipate that this overhead can be lowered significantly.

Further, we can use cheaper alternatives to page protection for protecting the address map data-structure. For example, heap randomization techniques can be used to hide data structure copies as opposed to paying the cost of turning on hard protections at boundary crossings [3]. Alternatively, the address map structure may be hidden in a separate process, using a technique similar to the one proposed by Berger et al. [9]. These techniques would make

[4] Average of five timed executions on a machine running CentOS 5.4 on four dual-core 2.8 GHz AMD Opteron 8220s; 8Gb RAM.

[5] Average of five timed executions on a virtual machine running Ubuntu 9.10 on a 2.13Ghz Intel Core 2 Duo; 722Mb RAM.

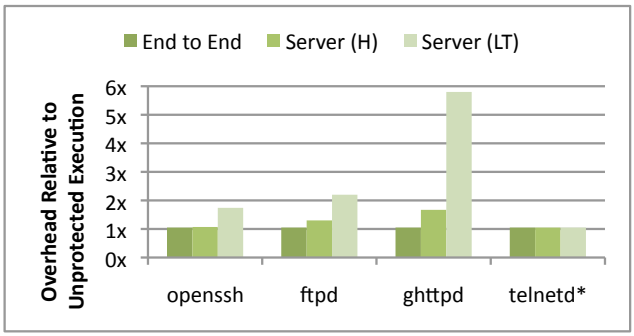

Figure 8. CPU overhead for hardening data vulnerabilities using YARRA's targeted protection mode, measured from the client ("End to End") and server perspectives. There was no measurable overhead from the client's perspective with either the hash table (H) or look-up table (LT) address maps. A value of *1x* indicates no measurable overhead.

boundary crossings take constant time (instead of being linear with the size of the map), albeit at the cost of look-up speed.

6. Related work

Array Bounds Checking. Early array bounds-checking techniques (e.g., Jones and Lin [11]) had substantial performance overheads, and more recent work (e.g., [16] as a recent example) attempts to reduce that overhead. Approaches to memory safety through array bounds checking fail to provide complete safety unless every memory reference is checked, including references from modules that have not been compiled with checking enabled. YARRA differs from this prior work in its emphasis on protecting the contents of arrays from all references made to *other objects*, including references made in arbitrary external libraries.

As mentioned, YARRA's explicit declaration of types has similarities to ideas in WIT [2]. Unlike WIT, YARRA allows the user to specify object equivalence classes explicity and precisely, and guarantees that all program references, including those performed in external components, do not violate the integrity of such objects.

None of the prior work on array bounds checking attempts to define the semantics of programs in which only some array bounds are checked. Dhurjati et al. [8] show that using a pool-allocation transformation, they are able to eliminate bounds checks altogether and ensure semantic correctness of array references even in the presence of incorrect frees. However, like other array bounds checking research, they assume that all code in an application has been transformed to ensure safety.

Separating and Isolating Memory. Software fault isolation [24] attempts to isolate the potential negative effects of external components by preventing memory operations and other unwanted interactons, such as system calls, that might be harmful. Castro et al. describe BGI (Byte-Granularity Isolation) [4], which provides software enforced protection domains between kernel extensions. Like YARRA, they provide an API that allows users to explicitly identify what extensions can access what memory. Unlike YARRA, BGI assumes that all untrusted extensions are compiled with BGI and will fail in the presence of untrusted extensions. In addition, unlike YARRA, BGI has no formal semantics.

Samurai [20] also takes the approach of explicitly protecting part of the entire memory state. Samurai focuses on making applications more fault tolerant in the presence of targeted runtime errors. Unlike Samurai, YARRA provides a precise definition of what critical memory means, incorporates those semantics in language features, and demonstrates that such features are useful to ensure correctness and security properties.

Formal Reasoning. The most closely related theories eminate from a line of research started in the 70s with the Euclid programming language [15]. Euclid was built in order to facilitate verification and one of the techniques for doing so involved logically, as opposed to physically, splitting the heap into a set of different heaplets called collections. These collections resemble the typed heaplets in this paper except that there was no means for moving an object from one heap to another as we do with bless and unbless operations. In the mid-nineties, Utting [23] reexamined Euclid's model and added a transfer coercion that, logically speaking, moved objects between heaplets, though physically, no action was taken. Recently, similar ideas have been rediscovered by Lahiri *et al.* [14]. They modernized and extended Euclid's Hoare Logic and illustrated the interaction between collections, now called *linear maps*, and the frame rule. The key difference between YARRA and this previous work is that YARRA's separate heaplets are designed to be used in the context of an unsafe language with unverified libraries. Consequently, the bless and unbless operations (*i.e.,* transfers) have operational significance: they put up and tear down physical protections.

7. Conclusions

This paper has presented YARRA, a new, lightweight extension to C. Using a combination of techniques, including lightweight type-based specifications and efficient runtime monitoring, YARRA allows programmers to protect the integrity of critical data structures in their programs, even in the presence of untrusted third-party libraries. A key contribution of our work is that we have been able to define a formal model for YARRA that can represent these untrusted, possibly buggy libraries and yet, through our powerful type-based frame rule, still allow programmers to reason soundly and modularly about core components of their programs. In addition, in practice, we have shown YARRA to be effective in protecting important server applications from known vulnerabilities. We were able to harden applications tens of thousands of lines long by modifying at most a few hundred lines of code of each application. Moreover, the end-to-end performance overhead was negligible in the security-centric examples we studied. Consequently, YARRA represents a viable new technology that complements existing safety mechanisms for C programs.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *CCS*, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE S&P*, 2008.

[3] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[4] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security (SSYM)*, 2005.

[6] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security (SSYM)*, 1998.

[7] DEP: Data execution prevention. http://support.microsoft.com/kb/875352.

[8] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. *SIGPLAN Not.*, 38(7):69–80, 2003.

[9] T. L. Emery D. Berger, Ting Yang and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX (ATC)*, 2002.

[11] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, 1997.

[12] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *ASPLOS*, 2006.

[13] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, 2008.

[14] S. Lahiri, S. Qadeer, and D. Walker. Linear maps. In *PLPV*, 2011.

[15] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12(2), 1977.

[16] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI*, 2009.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, 2002.

[18] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL*, 2002.

[19] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[20] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: protecting critical data in unsafe languages. *SIGOPS Oper. Syst. Rev.*, 2008.

[21] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[22] A. Sotirov. Modern exploitation and memory protection bypasses. http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf, 2009.

[23] M. Utting. Reasoning about aliasing. In *Fourth Australasian Refinement Workshop*, pages 195–211, 1995.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, pages 203–216, 1993.

[25] J. Walker. The BGET memory allocator. http://www.fourmilab.ch/bget/, 1996.

[26] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *SRDS*, 2003.

A. Full semantics and soundness of YCORE

The full semantics of YCORE is presented in a series of figures starting on page 16. A proof of YCORE's soundness follows.

Lemma 2 (Substitution (intrepretation of formulas))**.**
1. *For all* $E, E', x, i, \Phi, \Gamma_1, \Gamma_2$ *such that* $\vdash (E, x \mapsto i, E') : \Gamma_1, x, \Gamma_2$*; and* $\Gamma_1, x, \Gamma_2 \vdash \Phi$ *ok:* $E, x \mapsto i, E' \models \Phi \iff E, E' \models \Phi[i/x]$.
2. *For all* $E, E', X, \hat{v}, \hat{\tau}, \Phi, \Gamma_1, \Gamma_2$ *such that* $\vdash (E, X \mapsto (\hat{v}{:}\hat{\tau}), E') : \Gamma_1, X{:}\hat{\tau}, \Gamma_2$*; and* $\Gamma_1, X{:}\hat{\tau}, \Gamma_2 \vdash \Phi$ *ok:* $E, X \mapsto (\hat{v}{:}\hat{\tau}), E' \models \Phi \iff E, E' \models \Phi[\hat{v}/X]$.

Proof. By induction on the structure of the denotation for formula entailment. □

Lemma 3 (Monotonicity of equality)**.**
For all $E, \Gamma, \Phi, x, \tau, a_1, a_2$ *such that* $\vdash E : \Gamma$*; and* $\Gamma, x{:}\tau \vdash \Phi$ *ok; and* $\Gamma \vdash a_1 : \tau$*; and* $\Gamma \vdash a_2 : \tau$*; and* $E \vdash a_1 = a_2$*:* $E \models \Phi[a_1/x] \Rightarrow E \models \Phi[a_2/x]$

Proof. By induction on the structure of the denotation for formula entailment. □

Lemma 4 (Weakening and guarded contraction (intrepretation of formulas))**.**
1. *For all* $E_1, E_2, x, i, \Gamma_1, \Gamma_2, \Phi$*, such that* $\vdash E_1, x \mapsto i, E_2 : \Gamma_1, x, \Gamma_2$*; and* $\vdash E_1, E_2 : \Gamma_1, \Gamma_2$*; and* $\Gamma_1, \Gamma_2 \vdash \Phi$ *ok:* $E_1, E_2 \models \Phi \iff E_1, x \mapsto i, E_2 \models \Phi$.
2. *For all* $E_1, E_2, X, \hat{v}, \hat{\tau}, \Gamma_1, \Gamma_2, \Phi$*, such that* $\vdash E_1, X \mapsto (\hat{v}{:}\hat{\tau}), E_2 : \Gamma_1, X{:}\hat{\tau}, \Gamma_2$*; and* $\vdash E_1, E_2 : \Gamma_1, \Gamma_2$*; and* $\Gamma_1, \Gamma_2 \vdash \Phi$ *ok:* $E_1, E_2 \models \Phi \iff E_1, X \mapsto (\hat{v}{:}\hat{\tau}), E_2 \models \Phi$.

Proof. By induction on the structure of the denotation for formula entailment. □

Corollary 5 (Moving substitutions)**.**
For all $E, X, \hat{v}, \Gamma, \Phi, a$ *such that* $\vdash E : \Gamma$*; and* $\vdash E[X \leftarrow \hat{v}] : \Gamma$*; and* $\Gamma \vdash \Phi$ *ok; and* $\Gamma \vdash a = \hat{v}$ *ok; and* $E \models a = \hat{v}$*:* $E[X \leftarrow \hat{v}] \models \Phi \iff E \models \Phi[a/X]$.

Proof. Corollary of Lemmas 2, 3, and 4. □

Lemma 6 (Well-typed auxiliary functions)**.**
1. *For all* $E, \Gamma, \tau, \ell, E'$*:* $\vdash E : \Gamma \wedge \Gamma \vdash \tau$ *ok* $\wedge$ *copy*$_E$ ℓ *from* H *to* $\tau = E' \Rightarrow \vdash E' : \Gamma$
2. *For all* $E, \Gamma, \tau, \ell, E'$*:* $\vdash E : \Gamma \wedge \Gamma \vdash \tau$ *ok* $\wedge$ *chkAndRem*$_E$ τ $\ell = E' \Rightarrow \vdash E' : \Gamma$
3. *For all* E, Γ, x, ℓ*:* $\vdash E : \Gamma \wedge \Gamma \vdash x$ *ok* $\Rightarrow \vdash E[x \mapsto \ell] : \Gamma$
4. *For all* E, Γ, τ, a_L, e*:* $\vdash E : \Gamma \wedge \Gamma \vdash \tau$ *ok* $\wedge \Gamma \vdash a_L :$ *int set* $\wedge \Gamma \vdash e :$ *int* $\wedge$ *updUn*$_E$ a_L τ $e = E' \Rightarrow \vdash E' : \Gamma$

Proof. By inspection. □

Lemma 7 (Relating static and dynamic: *readFrom*)**.** *For all* $E, \Gamma, X, \hat{\tau}_X, \tau, \ell$ *such that* $\vdash E : \Gamma$*; and* $\Gamma \vdash X : \hat{\tau}_X$*; and* $\Gamma \vdash \tau$ *ok:* $E \models$ *readFrom*$_E$ X $(\ell{:}\tau)$ = *readFrom*$_\Gamma$ X $(\ell{:}\tau)$

Proof. By induction on the shape of τ, observing that $\forall E, Y, \ell'.E \vdash Y_E(\ell') = Y_\Gamma(\ell')$. □

Lemma 8 (Relating static and dynamic: *copyInto*)**.**
For all $E, \Gamma, X, \hat{\tau}, \tau, a_L$ *such that* $\vdash E : \Gamma$*; and* $\Gamma \vdash \tau$ *ok; and* $\Gamma \vdash X : \hat{\tau}$*; and* $\Gamma \vdash a_L :$ *int set:*

1. $\forall X_1, \ldots, X_n, \hat{v}_1, \ldots, \hat{v}_n$. *if copy*$_E$ a_L *from* X *to* $\tau = E[X_1 \leftarrow \hat{v}_1] \ldots [X_n \leftarrow \hat{v}_n]$ *then* $\exists a_1, \ldots, a_n$ *such that copy*$_\Gamma$ a_L *from* X *to* $\tau = [a_1/X_1], \ldots, [a_n/X_n]$ *and* $\forall 1 \leq i \leq n.E \models a_i = \hat{v}_i \wedge (X_i = X \vee X_i \in \tau)$.
2. $\forall X_1, \ldots, X_n, a_1, \ldots, a_n$. *if copy*$_\Gamma$ a_L *from* X *to* $\tau = [a_1/X_1], \ldots, [a_n/X_n]$ *then* $\exists \hat{v}_1, \ldots, \hat{v}_n$ *such that copy*$_E$ a_L *from* X *to* $\tau = E[X_1 \leftarrow \hat{v}_1] \ldots [X_n \leftarrow \hat{v}_n]$ *and* $\forall 1 \leq i \leq n.E \models a_i = \hat{v}_i$.

Proof. By induction on the structure of τ, appealing to Lemma 7. □

Lemma 9 (Relating static and dynamic: *chkAndRem*)**.**
For all E, Γ, τ, a_L*, such that* $\vdash E : \Gamma$*; and* $\Gamma \vdash \tau$ *ok; and* $\Gamma \vdash a_L :$ *int set:*

1. $\forall X_1, \ldots, X_n, \hat{v}_1, \ldots, \hat{v}_n$. *if chkAndRem*$_E$ τ $a_L = E[X_1 \leftarrow \hat{v}_1] \ldots [X_n \leftarrow \hat{v}_n]$ *then* $\exists \Phi, a_1, \ldots, a_n$. *such that chkAndRem*$_\Gamma$ τ $a_L = \Phi, [a_1/X_1] \ldots [a_n/X_n]$ *and* $E \models \Phi$ *and* $\forall 1 \leq i \leq n.E \models a_i = \hat{v}_i \wedge X_i \in \tau$.
2. $\forall X_1, \ldots, X_n, a_1, \ldots, a_n, \Phi$. *if chkAndRem*$_\Gamma$ τ $a_L = \Phi, [a_1/X_1] \ldots [a_n/X_n]$ *and* $E \models \Phi$ *then either* (*chkAndRem*$_E$ τ $a_L = notSync$) *or* $\exists \hat{v}_1, \ldots, \hat{v}_n$. *such that* (*chkAndRem*$_E$ τ $a_L = E[X_1 \leftarrow \hat{v}_1] \ldots [X_n \leftarrow \hat{v}_n]$ *and* $\forall i.E \models a_i = \hat{v}_i$.

Proof. By induction on the structure of τ. □

Lemma 10 (Relating static and dynamic: *updUn*)**.**
For all E, Γ, τ, a_L, e*, such that* $\vdash E : \Gamma$*; and* $\Gamma \vdash \tau$ *ok; and* $\Gamma \vdash e :$ *int:*

1. *For all* $\hat{v}$ *such that updUn*$_E$ a_L τ $i = E[Un \leftarrow \hat{v}]$ *there exists* a *such that updUn*$_\Gamma$ a_L τ $= [a/Un]$ *and* $E \models a = \hat{v}$.
2. *For all* a *such that updUn*$_\Gamma$ a_L τ $= [a/Un]$ *there exists* $\hat{v}$ *such that updUn*$_E$ a_L τ $i = E[Un \leftarrow \hat{v}]$ *and* $E \vdash a = \hat{v}$.

Proof. By induction on the structure of τ. □

Lemma 11 (Framing (interpretation of formulas))**.**
For all $E_1, E_2, \Gamma_1, \Gamma_2, \Phi$ *such that* $\vdash E_1 : \Gamma_1$*; and* $\vdash E_2 : \Gamma_1, \Gamma_2$*; and* $\Gamma_1 \vdash \Phi$ *ok; and* $\forall x, X \in FV(\Phi).E_1(x) = E_2(x) \wedge E_1(X) = E_2(X)$*:*
$E_1 \models \Phi \iff E_2 \models \Phi$.

Proof. By induction over the structure of the denotation of formulas. □

Theorem 12 (Preservation)**.** *For all programs* s, s'*, environments* Γ*,* Δ*, formulas* Φ, Ψ*, and stores* E, E'*; if all of the following hold true:*

(H1) $\vdash \Gamma; \Delta$ *ok, i.e., the verification environment is well-formed*
(H2) $\vdash E : \Gamma$*, i.e., the store* E *is well-formed according to* Γ
(H3) $\Gamma \vdash \Psi$ *ok, i.e., the post-condition is well-formed in* Γ
(H4) $\Gamma; \Delta \vdash \{\Phi\}\ s\ \{\Psi\}$*, i.e., the program* s *is verifiable with pre-condition* Φ
(H5) $E \models \Phi$*, i.e., the pre-condition is satisfiable in the store* E
(H6) $(E; s) \rightsquigarrow (E'; s')$*, i.e., the program takes a single step*

Then, all of the following are valid:

(G0) For all x, X*,* $x \notin \Delta \Rightarrow E(x) = E'(x)$ *and* $X \notin \Delta \Rightarrow E(X) = E'(X)$.
(G1) There exists Γ'*, such that* $\vdash E' : \Gamma'$*, i.e.,* E' *is well-formed according to* Γ'.
(G2) There exists Γ'' *such that* $\Gamma' = \Gamma, \Gamma''$*; i.e.,* Γ' *is an extension of* Γ.
(G3) There exists Φ' *such that* $E' \models \Phi'$.
(G4) $\Gamma'; \Delta' \vdash \{\Phi'\}\ s'\ \{\Psi\}$*, where* $\Delta' = \Delta \cup (dom\ \Gamma')$.

Proof. By induction on the structure of the verification derivation, hypothesis (H4).

Case T-Frame:
(H4) is $\dfrac{\Gamma; \Delta \setminus FV(\Phi_1) \vdash \{\Phi_2\}\ s\ \{\Psi\}}{\Gamma; \Delta \vdash \{\Phi_1 \wedge \Phi_2\}\ s\ \{\Phi' \wedge \Psi\}}$ and from (H5) we have $E \models \Phi_1 \wedge \Phi_2$ and hence (H5.1) $E \models \Phi_1$ and (H5.2) $E \models \Phi_2$.
From the induction hypothesis applied to the first premise of (H4), we obtain
(G0') $E'(X) = E(X)$ for all $X \in FV(\Phi)$ (likewise for $x \in FV(\Phi)$)
(G1') and (G2') $\vdash E' : \Gamma'$ and $\Gamma' = \Gamma, \Gamma''$
(G3') $E' \models \Phi_2'$ and (G4') $\Gamma'; \Delta' \vdash \{\Phi_2'\}\ s'\ \{\Psi\}$
For the goals, we obtain (G0), (G1), and (G2) immediately from (G0'), (G1') and (G2').
For (G3), we show that $E' \models \Phi_1 \wedge \Phi_2'$ from (G3') and from Lemma 11 applied to (G0') and (H5.1).
For (G4), we derive $\Gamma'; \Delta' \vdash \{\Phi_1 \wedge \Phi_2'\}\ s'\ \{\Psi\}$ using (T-Frame) with (G4') in the premise, noting that $dom\ \Gamma' \cap FV(\Phi_1) = \emptyset$.

Case T-Loc:
(H4) is $\dfrac{x \notin dom\ \Gamma \qquad \Gamma, x; \Delta, x \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma; \Delta \vdash \{\forall x.\Phi\}\ \text{local } x \text{ in } s\ \{\Psi\}}$ and, from (H5), we have $E \models \forall x.\Phi$.

By inversion on the reduction relation, we have (H6) $(E; \text{local } x \text{ in } s) \rightsquigarrow (E'; s)$, with $E' = E, x \mapsto i$ for arbitrary i.
For (G0), is immediate by noting that E' is an extension of E.
For (G1) and (G2): we have $\vdash E' : \Gamma, x$, by an extension of (H2), appealing to α-conversion for $x \notin dom\ E$.
For (G3), we show $E, x \mapsto i \models \Phi$, we use Lemma 2 and show $E \models \Phi[i/x]$ since from (H5), we have $E \models \Phi[j/x]$, for any j.
Finally, for (G4), we invert (H4) and use its second premise.

Case T-NewX:
(H4) is $\dfrac{\Gamma \vdash \tau \text{ ok} \qquad X \notin dom\ \Gamma \qquad \hat{\tau} = int \to \tau \qquad \Gamma, X{:}\hat{\tau}; \Delta, X \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma; \Delta \vdash \{\forall X{:}\hat{\tau}.X = \lambda\ell.\bot \Rightarrow \Phi\}\ \text{newtype } X = \tau \text{ in } s\ \{\Psi\}}$

By inversion on the reduction relation, we have $(E; \text{newtype } X = \tau \text{ in } s) \rightsquigarrow (E, X \mapsto (\lambda\ell.\bot{:}int \to \tau); s)$
The goals follow as in case (T-Loc), while observing that the X is specifically the empty map, to satisfy the implication guard.

Case T-Bless:
(H4) is $\dfrac{\begin{array}{c}\Gamma \vdash e_1, e_2, y \text{ ok} \qquad L = \bigcup_{0 \le i < e_1} \{e_2 + |X|_\Gamma * i\} \qquad range_\Gamma\ X = \tau \qquad y, X, Un, \tau \in \Delta \\ \sigma_1 = copy_\Gamma\ L \text{ from } H \text{ to } X \qquad \Phi, \sigma_2 = chkAndRem_\Gamma\ \tau\ L \qquad \sigma_3 = updUn_\Gamma\ L\ \tau\ \bot\end{array}}{\Gamma; \Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := \text{bless}_X[e_1]\ e_2\ \{\Psi\}}$

Inversion of (H6) gives one of two possible applications of (E-Bless), resulting in two sub-cases as below:

Sub-case (E-Bless-Abort): Goals are trivial, using an application of (T-Abort)

Sub-case (E-Bless):

$$\text{(H6) is } \frac{[\![e_1]\!]_E = n \qquad [\![e_2]\!]_E = \ell \qquad L = \bigcup_{0 \le i < n} \{\ell + |X|_E * i\} \qquad \tau = \mathit{range}_E\ X \qquad E_1 = \mathit{chkAndRem}_E\ \tau\ L \qquad E_2 = \mathit{copy}_{E_1}\ L\ \mathit{from}\ H\ \mathit{to}\ X \qquad E' = \mathit{updUn}_{E_2}\ L\ \tau\ \bot}{(E; y := \text{bless}_X[e_1]\ e_2) \rightsquigarrow (E'[y \mapsto \ell]; \text{skip})}$$

For goal (G0), we appeal to Lemmas 8, 9, and 10 to conclude that $E' = E[\mathit{Un} \leftarrow \hat{v}_1][X \leftarrow \hat{v}_2][X_3 \leftarrow \hat{v}_3] \ldots [X_n \leftarrow \hat{v}_n]$, where $X_3, \ldots X_n \in \tau$.
From the premises of (H6), we have $X, \mathit{Un}, y, \tau \in \Delta$, sufficient to establish (G0), since $E'[y \mapsto \ell]$ differs from E only on the said locations.
For goals (G1) and (G2): we pick $\Gamma' = \Gamma$, and from Lemma 6, we get $\vdash E' : \Gamma$.
For (G3) and (G4), we pick $\Phi' = \Psi$ and apply (T-Skip) to get $\Gamma; \Delta \vdash \{\Psi\}$ skip $\{\Psi\}$
It remains to be shown that $E' \models \Psi$.
From (H5), we know $E \models \Phi \wedge \sigma(\Psi)$ and hence $E \models \sigma(\Psi)$
From Lemmas 8, 9, and 10 we get that $\sigma = [a_1/\mathit{Un}][a_2/X][a_3/X_1] \ldots [a_n/X_n][e_2/y]$ with $\forall i. E \models a_i = \hat{v}_i$
From repeated application of Corollary 5, we get $E' \models \Psi$, as required.

Case T-UnBless: Analogous to T-Bless.

Case T-Write:

$$\text{We have (H4) } \frac{\Gamma \vdash e_1, e_2 \text{ ok} \qquad X, H \in \Delta \qquad X \neq \mathit{Un} \qquad v_h = \mathit{readFrom}_\Gamma\ H\ (e_1{:}X) \qquad v_x = X_\Gamma(e_1) \qquad H_1 = H[(e_1 + \mathit{offset}_\Gamma\ X\ p) \leftarrow e_2] \qquad \sigma_1 = \mathit{copy}_\Gamma\ e_1\ \mathit{from}\ H_1\ \mathit{to}\ X \qquad \sigma = \sigma_1 \circ [H_1/H]}{\Gamma; \Delta \vdash \{e_1 \in \mathit{dom}_\Gamma\ X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ X(e_1).p := e_2\ \{\Psi\}}$$

Inverting (H6) we get one of two sub-cases:
Sub-case E-Write-Abort: Trivial.
Sub-case E-Write:

$$\text{We have (H6) } \frac{p \neq \cdot \qquad [\![e_1]\!]_E = \ell \qquad [\![e_2]\!]_E = v \qquad \ell \in \mathit{dom}_\mathcal{E}\ X \qquad \mathit{inSync}_E\ \ell\ X \qquad \ell' = \ell + \mathit{offset}_E\ X\ p \qquad E_1 = E[H \leftarrow (\ell' \mapsto v)] \qquad E' = \mathit{copy}_{E_1}\ \{\ell\}\ \mathit{from}\ H\ \mathit{to}\ X}{(E; X(e_1).p := e_2) \rightsquigarrow (E'; \text{skip})}$$

For (G0), we use Lemma 8 to observe that $E' = E[H \leftarrow \hat{v}_1][X \leftarrow \hat{v}_2]$, and note that both $X, H \in \Delta$.
For (G1) and (G2), we show that $\vdash E' : \Gamma$, using Lemma 6.
For (G4), we derive $\Gamma; \Delta \vdash \{\Psi\}$ skip $\{\Psi\}$ using T-Skip.
For (G3), we need to show $E' \models \Psi$.
From (H5) we have $E \models (\mathit{readFrom}_\Gamma\ H\ (e_1{:}X) = X_\Gamma(e_1)) \Rightarrow \sigma(\Psi)$.
From the premises of (H6) we have $\mathit{inSync}_E\ \ell\ X$, from which we obtain $X = \mathit{Un}$ or $E \models X_E(\ell) = \mathit{readFrom}_E\ H\ (e_1{:}X)$.
From the premises of (H4) we have $X \neq \mathit{Un}$.
Thus, we have $E \models \sigma(\Psi)$, and we use Corollary 5 to get $E' \models \Psi$, as required.

Case T-Read: Similar to (T-Write).

Case T-IsX:

$$\text{We have (H4) } \frac{\Gamma \vdash e \text{ ok} \qquad v_h = \mathit{readFrom}_\Gamma\ H\ (e{:}X) \qquad v_x = X_\Gamma(e) \qquad \Gamma; \Delta \vdash \{\Phi_1\}\ s_1\ \{\Psi\} \qquad \Gamma; \Delta \vdash \{\Phi_2\}\ s_2\ \{\Psi\}}{\Gamma; \Delta \vdash \{((e \in \mathit{dom}_\Gamma\ X \wedge (X = \mathit{Un} \vee v_h = v_x)) \Rightarrow \Phi_1) \wedge (e \notin \mathit{dom}_\Gamma\ X \Rightarrow \Phi_2)\}\ \text{if } e \text{ is in } X \text{ then } s_1 \text{ else } s_2\ \{\Psi\}}$$

Inverting (H6) we get one of three sub-cases.
Sub-case E-IsX-Abort: Trivial.

Sub-case E-IsX-Then:

$$\text{(H6) is } \frac{[\![e]\!]_E = \ell \qquad \ell \in \mathit{dom}_\mathcal{E}\ X \qquad \mathit{inSync}_E\ \ell\ X}{(E; \text{if } e \text{ is in } X \text{ then } s_1 \text{ else } s_2) \rightsquigarrow (E; s_1)}$$

Goals (G0), (G1) and (G2) are trivial, since the store is unchanged.
For (G3) we show $E \models \Phi_1$, since we have $E \models (X = \mathit{Un} \vee v_h = v_x)) \Rightarrow \Phi_1$ from (H5).
From the premises of (H6) we have $\mathit{inSync}_E\ \ell\ X$, from which we obtain $X = \mathit{Un}$ or $E \models X_E(\ell) = \mathit{readFrom}_E\ H\ (e_1{:}X)$.
This suffices to show $E \models \Phi_1$.
For (G4), we use the premise of (H4) to show $\Gamma; \Delta \vdash \{\Phi_1\}\ s_1\ \{\Psi\}$.

Sub-case E-IsX-Else:

$$\text{(H6) is } \frac{[\![e]\!]_E = \ell \qquad \ell \notin \mathit{dom}_E\ X}{(E; \text{if } e \text{ is in } X \text{ then } s_1 \text{ else } s_2) \rightsquigarrow (E; s_2)}$$

Goals (G0), (G1) and (G2) are trivial, since the store is unchanged.
For (G3) we show $E \models \Phi_2$, since we have $E \models (e \notin dom_\Gamma\ X \Rightarrow \Phi_2)$ from (H5).
From the premises of (H6) we have $\ell \notin dom_E\ X$, which suffices.
For (G4), we use the premise of (H4) to show $\Gamma; \Delta \vdash \{\Phi_2\}\ s_2\ \{\Psi\}$.

Case T-LibWrite:

We have (H4) $\dfrac{\Gamma \vdash e_1, e_2 \text{ ok} \qquad H_1 = H[e_1 \leftarrow e_2] \qquad \sigma = [H_1/H]}{\Gamma; H \vdash \{\sigma(\Psi)\}\ \text{lib}\ e_1 := e_2\ \{\Psi\}}$

Inverting (H6), we get $\dfrac{[\![e_1]\!]_E = \ell \qquad [\![e_2]\!]_E = v \qquad E' = E[H \leftarrow (\ell \mapsto v)]}{(E; \text{lib}\ e_1 := e_2) \rightsquigarrow (E'; \text{skip})}$

For (G0), we have $H \in \Delta$ and $E' = E[H \leftarrow \ldots]$.

For (G1) and (G2) we have $\vdash E' : \Gamma$, from $\vdash v : int$.

For (G3), we show $E' \models \Psi$, from $E \vdash \sigma(\Psi)$ and Corollary 5.
For (G4), we we use (T-Skip) for $\Gamma'; \Delta \vdash \{\Psi\}$ skip $\{\Psi\}$.

Case T-LibRead: Similar to T-LibWrite.

Case T-Cons, T-Assert, T-If, T-While, T-Seq, T-Skip, T-Abort: All standard.

□

Theorem 13 (Progress)**.** *For all programs s, environments Γ, Δ, formulas Φ, Ψ, and stores E; if all of the following hold true:*

(H1) $\vdash \Gamma; \Delta$ *ok, i.e., the verification environment is well-formed*
(H2) $\vdash E : \Gamma$*, i.e., the store E is well-formed according to Γ*
(H3) $\Gamma \vdash \Psi$ *ok, i.e., the post-condition is well-formed in Γ*
(H4) $\Gamma; \Delta \vdash \{\Phi\}\ s\ \{\Psi\}$*, i.e., the program s is verifiable with pre-condition Φ*
(H5) $E \models \Phi$*, i.e., the pre-condition is satisfiable in the store E*
(H6) $s \neq$ *skip.*

Then, there exists E', s' such that all of the following are valid:

(G1) $(E; s) \rightsquigarrow (E'; s')$*; i.e., the program can take a single step.*
(G2) If $\forall X, l.l \in dom\ X \Rightarrow inSync_E\ X\ l$ *then* $s' \neq$ *abort.*

Proof. Straightforward induction over the structure of the verification judgment (H4). □

meta-variables
  local variables x, y, z
  integer constants i, j, ℓ
  object type names/heaplet variables X, Y, Z, and distinguished names H for the heap and the *Un*-color
  predicate variables P, Q, R

expressions	e	::=	$i \mid x \mid e\ op\ e'$	
statements	s	::=	assert Φ	assert formula Φ
		$\mid$	newtype $X = \tau$ in s	new object declaration (scoped)
		$\mid$	$y :=$ bless$_X[e_1]\ e_2$	bless (optional array size)
		$\mid$	$y :=$ unbless$_X\ [e_1]\ e_2$	unbless (optional array size)
		$\mid$	if e is in X then s_1 else s_2	test membership in a heaplet ...
		$\mid$	local x in s	local ...
		$\mid$	$X(e_1).p := e_2$	
		$\mid$	lib $e_1 := e_2$	
		$\mid$	$y := X(e_1).p$	
		$\mid$	lib $y := e_1$	
		$\mid$	if e_1 then s_1 else s_2	
		$\mid$	while $e\ s \mid s_1; s_2 \mid$ skip	
		$\mid$	abort	
path	p	::=	$\cdot \mid 0p \mid 1p$	field projection path
types	τ	::=	$int \mid X \mid (\tau_1, \tau_2)$	
values	v	::=	$i \mid (v_1, v_2)$	
map types	$\hat{\tau}$	::=	$int \to \tau$	
map values	$\hat{v}$	::=	$\lambda\ell.\hat{e}$	
map body	$\hat{e}$	::=	$\bot \mid v \mid \hat{v}\ v \mid$ if $a \in a'$ then $\hat{e}$ else $\hat{e}'$	
logic terms	a	::=	$e \mid v \mid X \mid \hat{v} \mid a.p \mid \hat{e} \mid dom\ a \mid \{x \mid \Phi\}$	
formulas	Φ, Ψ	::=	$a = a' \mid a \in a' \mid a < a' \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \neg\Phi \mid \forall x.\Phi \mid \exists x.\Phi$	
substitutions	σ	::=	$\sigma, [a/X] \mid \sigma, [a/x] \mid \cdot$	
runtime env.	E	::=	$H \mapsto (\hat{v}{:}\hat{\tau}), Un \mapsto (\hat{v}{:}\hat{\tau}) \mid E, X \mapsto (\hat{v}{:}\hat{\tau}) \mid E, x \mapsto i$	
static env.	Γ	::=	$H{:}\hat{\tau}, Un{:}\hat{\tau} \mid \Gamma, X{:}\hat{\tau} \mid \Gamma, x$	
environment	$\mathcal{E}$	::=	$E \mid \Gamma$	

Figure 9. Syntax

Many of these functions are overloaded to operate on both static environments Γ and runtime stores E. We use $\mathcal{E} ::= \Gamma \mid E$.

$X_E(\ell)$	=	v	**when** $E(X) = (\hat{v}{:}\tau)$ and $\hat{v}\ \ell \rightsquigarrow v$
$X_\Gamma(\ell)$	=	$X\ \ell$	
$dom_E\ X$	=	$\{\ell \mid X_\mathcal{E}(\ell) \neq \bot\}$	
$dom_\Gamma\ X$	=	$dom\ X$	
$range_E\ X$	=	τ	**when** $E(X) = (\hat{v}{:}int \to \tau)$
$range_\Gamma\ X$	=	τ	**when** $\Gamma(X) = int \to \tau$
$\lvert int\rvert_\mathcal{E}$	=	1	
$\lvert Y\rvert_\mathcal{E}$	=	$\lvert range_\mathcal{E}\ Y\rvert_\mathcal{E}$	
$\lvert(\tau_1, \tau_2)\rvert_\mathcal{E}$	=	$\lvert\tau_1\rvert_\mathcal{E} + \lvert\tau_2\rvert_\mathcal{E}$	
$offset_\mathcal{E}\ \tau\ \cdot$	=	0	
$offset_\mathcal{E}\ (\tau_1, \tau_2)\ 0p$	=	$offset_\mathcal{E}\ \tau_1\ p$	
$offset_\mathcal{E}\ (\tau_1, \tau_2)\ 1p$	=	$\lvert\tau_1\rvert_\mathcal{E} + offset_\mathcal{E}\ \tau_2\ p$	
$offset_\mathcal{E}\ Y\ p$	=	$offset_\mathcal{E}\ (range_\mathcal{E}\ Y)\ p$	
$readFrom_\mathcal{E}\ Y\ (\ell{:}int)$	=	$Y_\mathcal{E}(\ell)$	
$readFrom_\mathcal{E}\ Y\ (\ell{:}Z)$	=	$readFrom_\mathcal{E}\ Y\ (\ell{:}(range_\mathcal{E}\ Z))$	
$readFrom_\mathcal{E}\ Y\ (\ell{:}(\tau_1, \tau_2))$	=	$(readFrom_\mathcal{E}\ Y\ (\ell{:}\tau_1), readFrom_\mathcal{E}\ Y\ ((\ell + \lvert\tau_1\rvert_E){:}\tau_2))$	
$notBlessed_\mathcal{E}\ \ell$	=	$\ell \in dom_\mathcal{E}\ Un$	

Figure 10. Auxiliary functions used in both static and dynamic semantics

$$
\begin{array}{lcl}
X[a \leftarrow a'] & = & \lambda\ell.\text{if } \ell \in \{a\} \text{ then } a' \text{ else } (X\ell) \\
\{a\} & = & \{x \mid x = a\} \\
\bigcup_{a_1 \le i < a_2} \{x \mid \Phi\} & = & \{x \mid \exists i.(a_1 \le i < a_2 \wedge \Phi)\}
\end{array}
$$

copy-from-to : $(Env * Locs * Map * Type) \to Subst$

$$
\begin{array}{lcl}
\mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ \mathit{int} & = & \cdot \\
\mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ X & = & [(\lambda\ell.\text{if } \ell \in L \text{ then } (\mathit{readFrom}_\Gamma\ Y\ (\ell{:}X)) \text{ else } X\ \ell)/X] \\
\mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ (\tau_1, \tau_2) & = & \text{let } \sigma_1 = \mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ \tau_1 \text{ in} \\
 & & \text{let } \sigma_2 = \mathit{copy}_\Gamma\ \{\ell + |\tau_1|_\Gamma \mid \ell \in L\}\ \mathit{from}\ Y\ \mathit{to}\ \tau_2 \text{ in} \\
 & & \sigma_1 \circ \sigma_2
\end{array}
$$

chkAndRem : $(Env * Type * Locs) \to (Prop * Subst)$

$$
\begin{array}{lcl}
\mathit{chkAndRem}_\Gamma\ \mathit{int}\ L & = & (L \subseteq \mathit{dom}\ \mathit{Un}, \cdot) \\
\mathit{chkAndRem}_\Gamma\ X\ L & = & \text{let } \Phi = \forall x.x \in L \Rightarrow x \in \mathit{dom}_\Gamma\ (X) \text{ in} \\
 & & (\Phi, [(\lambda\ell.\text{if } \ell \in L \text{ then } \bot \text{ else } X\ \ell)/X]) \\
\mathit{chkAndRem}_\Gamma\ (\tau_1, \tau_2)\ L & = & \text{let } \Phi_1, \sigma_1 = \mathit{chkAndRem}_\Gamma\ \tau_1\ L \text{ in} \\
 & & \text{let } \Phi_2, \sigma_2 = \mathit{chkAndRem}_\Gamma\ \tau_1\ \{\ell + |\tau_1|_\Gamma \mid \ell \in L\} \text{ in} \\
 & & (\Phi_1 \wedge \Phi_2, \sigma_1 \circ \sigma_2)
\end{array}
$$

Membership of types in Δ and of map variables in types

$$
\begin{array}{lcl}
\mathit{int} \in \Delta & = & \mathit{True} \\
X \in \Delta & = & \exists \Delta_1, \Delta_2.\Delta = \Delta_1, X, \Delta_2 \\
(\tau_1, \tau_2) \in \Delta & = & \tau_1 \in \Delta \wedge \tau_2 \in \Delta \\
X \in \mathit{int} & = & \mathit{False} \\
X \in X & = & \mathit{True} \\
X \in (\tau_1, \tau_2) & = & X \in \tau_1 \vee X \in \tau_2
\end{array}
$$

updUn : $(Env * Locs * Type * MapBody) \to Subst$

$$
\begin{array}{lcl}
\mathit{updUn}_\Gamma\ L\ \mathit{int}\ \hat{e} & = & [\lambda\ell.\text{if } \ell \in L \text{ then } \hat{e} \text{ else } \mathit{Un}\ \ell/\mathit{Un}] \\
\mathit{updUn}_\Gamma\ L\ X\ \hat{e} & = & \cdot \\
\mathit{updUn}_\Gamma\ L\ (\tau_1, \tau_2)\ \hat{e} & = & \text{let } \sigma_1 = \mathit{updUn}_\Gamma\ L\ \tau_1\ \hat{e} \text{ in} \\
 & & \text{let } L_1 = \{\ell + |\tau_1|_\Gamma \mid \ell \in L\} \text{ in} \\
 & & \mathit{updUn}_\Gamma\ L_1\ \tau_2\ \hat{e}
\end{array}
$$

Figure 11. Auxiliary functions used in static semantics only (Reproduced from Figure 5)

$$
\begin{array}{lcll}
E[X \leftarrow \hat{v}] & = & E_1, X \mapsto (\hat{v}{:}\hat{\tau}), E_2 & \textbf{when } E = E_1, X \mapsto (\hat{v}{:}\hat{\tau}), E_2 \\
\mathit{blessed}_E\ L\ X & = & \forall \ell \in L.\ell \in \mathit{dom}_E\ X &
\end{array}
$$

inSync : $(Env * Loc * Map) \to Prop$

$$
\begin{array}{lcll}
\mathit{inSync}_E\ \ell\ \mathit{Un} & = & \mathit{True} & \\
\mathit{inSync}_E\ \ell\ X & = & X_E(\ell) = \mathit{readFrom}_E\ H\ (\ell{:}X) & \textbf{when } X \neq \mathit{Un} \\
\mathit{inSync}_E\ L\ X & = & \forall \ell \in L.X_E(\ell) = \mathit{readFrom}_E\ H\ (\ell{:}X) & \textbf{when } X \neq \mathit{Un}
\end{array}
$$

copy-from-to : $(Env * Locs * Map * Type) \to Env$

$$
\begin{array}{lcl}
\mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ \mathit{int} & = & E \\
\mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ X & = & E[X \leftarrow (\lambda\ell.\text{if } \ell \in L \text{ then } \mathit{readFrom}_E\ Y\ (\ell{:}X) \text{ else } X(\ell))] \\
\mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ (\tau_1, \tau_2) & = & \text{let } E_1 = \mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ \tau_1 \text{ in} \\
 & & \text{let } L_1 = \{\ell + |\tau_1|_{E_1} \mid \ell \in L\} \text{ in} \\
 & & \mathit{copy}_{E_1}\ L_1\ \mathit{from}\ Y\ \mathit{to}\ \tau_2
\end{array}
$$

chkAndRem : $(Env * Type * Locs) \rightharpoonup (Env \cup notSync)$ ***(partial function)***

$$
\begin{array}{lcll}
\mathit{chkAndRem}_E\ X\ L & = & notSync & \textbf{when } \mathit{blessed}_\mathcal{E}\ L\ X \wedge \neg \mathit{inSync}_E\ L\ X \\
\mathit{chkAndRem}_E\ X\ L & = & E[X \leftarrow \lambda\ell.\text{if } \ell \in L \text{ then } \bot \text{ else } (X\ell)] & \textbf{when } \mathit{blessed}_\mathcal{E}\ L\ X \wedge \mathit{inSync}_E\ L\ X \\
\mathit{chkAndRem}_E\ \mathit{int}\ L & = & E & \textbf{when } L \subseteq \mathit{dom}_E\ \mathit{Un} \\
\mathit{chkAndRem}_E\ (\tau_1, \tau_2)\ L & = & \text{let } E_1 = \mathit{chkAndRem}_E\ \tau_1\ L \text{ in} & \\
 & & \text{let } L_1 = \{\ell + |\tau_1|_{E_1} \mid \ell \in L\} \text{ in} & \\
 & & \mathit{chkAndRem}_{E_1}\ \tau_2\ L_1 &
\end{array}
$$

updUn : $(Env * Locs * Type * MapBody) \to (Env)$

$$
\begin{array}{lcl}
\mathit{updUn}_E\ L\ \mathit{int}\ \hat{e} & = & E[\mathit{Un} \leftarrow \lambda\ell.\text{if } \ell \in L \text{ then } \hat{e} \text{ else } \mathit{Un}\ \ell] \\
\mathit{updUn}_E\ L\ X\ \hat{e} & = & E \\
\mathit{updUn}_E\ L\ (\tau_1, \tau_2)\ \hat{e} & = & \text{let } E_1 = \mathit{updUn}_E\ L\ \tau_1\ \hat{e} \text{ in} \\
 & & \text{let } L_1 = \{\ell + |\tau_1|_{E_1} \mid \ell \in L\} \text{ in} \\
 & & \mathit{updUn}_{E_1}\ L_1\ \tau_2\ \hat{e}
\end{array}
$$

Figure 12. Auxiliary functions used in dynamic semantics only

$\boxed{(E; s) \rightsquigarrow (E'; s')}$ where $E ::= H \mapsto (\hat{v}{:}\hat{\tau}), \mathit{Un} \mapsto (\hat{v}{:}\hat{\tau}) \mid E, X \mapsto (\hat{v}{:}\hat{\tau}) \mid E, x \mapsto i$

$$\frac{\hat{\tau} = \mathit{int} \to \tau}{(E; \text{newtype } X = \tau \text{ in } s) \rightsquigarrow (E, X \mapsto (\lambda\ell.\bot{:}\hat{\tau}); s)}\ \text{E-NewType} \qquad \frac{}{(E; \text{local } x \text{ in } s) \rightsquigarrow (E, x \mapsto i; s)}\ \text{E-NewLoc}$$

$$\frac{\begin{array}{c}[\![e_1]\!]_E = n \quad [\![e_2]\!]_E = \ell \quad L = \bigcup_{0 \le i < n} \{\ell + |X|_E * i\} \quad \tau = \mathit{range}_E\ X \\ E_1 = \mathit{chkAndRem}_E\ \tau\ L \quad E_2 = \mathit{copy}_{E_1}\ L\ \mathit{from}\ H\ \mathit{to}\ X \quad E' = \mathit{updUn}_{E_2}\ L\ \tau\ \bot\end{array}}{(E; y := \text{bless}_X[e_1]\ e_2) \rightsquigarrow (E'[y \mapsto \ell]; \text{skip})}\ \text{E-Bless}$$

$$\frac{\begin{array}{c}[\![e_1]\!]_E = n \quad [\![e_2]\!]_E = \ell \quad L = \bigcup_{0 \le i < n} \{\ell + |X|_E * i\} \quad \tau = \mathit{range}_E\ X \\ E_1 = \mathit{chkAndRem}_E\ X\ L \quad E_2 = \mathit{copy}_{E_1}\ L\ \mathit{from}\ H\ \mathit{to}\ \tau \quad E' = \mathit{updUn}_{E_2}\ L\ \tau\ 1\end{array}}{(E; y := \text{unbless}_X\ [e_1]\ e_2) \rightsquigarrow (E'[y \mapsto \ell]; \text{skip})}\ \text{E-UnBless}$$

$$\frac{\begin{array}{c}[\![e_1]\!]_E = n \quad [\![e_2]\!]_E = \ell \quad L = \{\ell, \ldots, (\ell + |X|_{\mathcal{E}} * (n-1))\} \\ \tau = \mathit{range}_E\ X \quad \mathit{chkAndRem}_E\ \tau\ L\ =\ \mathit{notSync}\end{array}}{(E; y := \text{bless}_X[e_1]\ e_2) \rightsquigarrow (E; \text{abort})}\ \text{E-Bless-Abort}$$

$$\frac{\begin{array}{c}[\![e_1]\!]_E = n \quad [\![e_2]\!]_E = \ell \quad L = \{\ell, \ldots, (\ell + |X|_{\mathcal{E}} * (n-1))\} \\ \tau = \mathit{range}_E\ X \quad \mathit{chkAndRem}_E\ X\ L\ =\ \mathit{notSync}\end{array}}{(E; y := \text{unbless}_X\ [e_1]\ e_2) \rightsquigarrow (E; \text{abort})}\ \text{E-UnBless-Abort}$$

$$\frac{\begin{array}{c}p \neq \cdot \quad [\![e_1]\!]_E = \ell \quad [\![e_2]\!]_E = v \quad \ell \in \mathit{dom}_{\mathcal{E}}\ X \quad \mathit{inSync}_E\ \ell\ X \\ \ell' = \ell + \mathit{offset}_E\ X\ p \quad E_1 = E[H \leftarrow (\ell' \mapsto v)] \quad E' = \mathit{copy}_{E_1}\ \{\ell\}\ \mathit{from}\ H\ \mathit{to}\ X\end{array}}{(E; X(e_1).p := e_2) \rightsquigarrow (E'; \text{skip})}\ \text{E-Write}$$

$$\frac{\begin{array}{c}p \neq \cdot \quad [\![e_1]\!]_E = \ell \quad \ell \in \mathit{dom}_{\mathcal{E}}\ X \quad \mathit{inSync}_E\ \ell\ X \\ \ell' = \ell + \mathit{offset}_E\ X\ p \quad E' = E[y \mapsto H_E(\ell')]\end{array}}{(E; y := X(e_1).p) \rightsquigarrow (E'; \text{skip})}\ \text{E-Read}$$

$$\frac{[\![e_1]\!]_E = \ell \quad \ell \in \mathit{dom}_{\mathcal{E}}\ X \quad \neg\mathit{inSync}_E\ \ell\ X}{(E; X(e_1).p := e_2) \rightsquigarrow (E; \text{abort})}\ \text{E-Write-Abort} \qquad \frac{[\![e_1]\!]_E = \ell \quad \ell \in \mathit{dom}_{\mathcal{E}}\ X \quad \neg\mathit{inSync}_E\ \ell\ X}{(E; y := X(e_1).p) \rightsquigarrow (E; \text{abort})}\ \text{E-Read-Abort}$$

$$\frac{[\![e_1]\!]_E = \ell \quad E' = E[y \mapsto H_E(\ell)]}{(E; \text{lib } y := e_1) \rightsquigarrow (E'; \text{skip})}\ \text{E-LibRd} \qquad \frac{[\![e_1]\!]_E = \ell \quad [\![e_2]\!]_E = v \quad E' = E[H \leftarrow (\ell \mapsto v)]}{(E; \text{lib } e_1 := e_2) \rightsquigarrow (E'; \text{skip})}\ \text{E-LibWrt}$$

$$\frac{[\![e]\!]_E = \ell \quad \ell \in \mathit{dom}_{\mathcal{E}}\ X \quad \mathit{inSync}_E\ \ell\ X}{(E; \text{if } e \text{ is in } X \text{ then } s_1 \text{ else } s_2) \rightsquigarrow (E; s_1)}\ \text{E-IsX-Then} \qquad \frac{[\![e]\!]_E = \ell \quad \ell \notin \mathit{dom}_E\ X}{(E; \text{if } e \text{ is in } X \text{ then } s_1 \text{ else } s_2) \rightsquigarrow (E; s_2)}\ \text{E-IsX-Else}$$

$$\frac{[\![e]\!]_E = \ell \quad \ell \in \mathit{dom}_E\ X \quad \neg\mathit{inSync}_E\ \ell\ X}{(E; \text{if } e \text{ is in } X \text{ then } s_1 \text{ else } s_2 \rightsquigarrow (E; \text{abort})}\ \text{E-IsX-Abort} \qquad \frac{[\![e]\!]_E \neq 0}{(E; \text{if } e \text{ then } s_1 \text{ else } s_2) \rightsquigarrow (E; s_1)}$$

$$\frac{[\![e]\!]_E = 0}{(E; \text{if } e \text{ then } s_1 \text{ else } s_2) \rightsquigarrow (E; s_2)} \qquad \frac{[\![e]\!]_E = 0}{(E; \text{while } e\ s) \rightsquigarrow (E; \text{skip})} \qquad \frac{[\![e]\!]_E \neq 0}{(E; \text{while } e\ s) \rightsquigarrow (E; (s; \text{while } e\ s))}$$

$$\frac{(E; s_1) \rightsquigarrow (E'; s_1')}{(E; (s_1; s_2)) \rightsquigarrow (E; (s_1'; s_2))} \qquad \frac{}{(E; (\text{skip}; s_2)) \rightsquigarrow (E; s_2)} \qquad \frac{E \models \Phi}{(E; \text{assert } \Phi) \rightsquigarrow (E; \text{skip})} \qquad \frac{}{(E; \text{abort}) \rightsquigarrow (E; \text{abort})}$$

Figure 13. Dynamic semantics of YCORE

$\boxed{\Gamma;\Delta \vdash \{\Phi\}\ s\ \{\Psi\}}$ where the set of locations modified by s is $\Delta ::= \Delta, X \mid \Delta, x \mid \cdot$

$$\frac{\Gamma;\Delta \vdash \{\Phi'\}\ s\ \{\Psi'\} \quad \Gamma \models (\Phi \Rightarrow \Phi') \quad \Gamma \models (\Psi' \Rightarrow \Psi)}{\Gamma;\Delta \vdash \{\Phi\}\ s\ \{\Psi\}}\ \text{T-Cons} \qquad \frac{\Gamma;\Delta \setminus FV(\Phi') \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma;\Delta \vdash \{\Phi' \wedge \Phi\}\ s\ \{\Phi' \wedge \Psi\}}\ \text{T-Frame}$$

$$\frac{\Gamma \vdash \Phi\ \text{ok}}{\Gamma;\Delta \vdash \{\Phi \wedge \Psi\}\ \text{assert}\ \Phi\ \{\Psi\}}\ \text{T-Assert} \qquad \frac{x \notin dom\ \Gamma \quad \Gamma, x;\Delta, x \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma;\Delta \vdash \{\forall x.\Phi\}\ \text{local}\ x\ \text{in}\ s\ \{\Psi\}}\ \text{T-Loc}$$

$$\frac{\Gamma \vdash \tau\ \text{ok} \quad X \notin dom\ \Gamma \quad \hat{\tau} = int \to \tau \quad \Gamma, X{:}\hat{\tau};\Delta, X \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma;\Delta \vdash \{\forall X{:}\hat{\tau}.X = \lambda\ell.\bot \Rightarrow \Phi\}\ \text{newtype}\ X = \tau\ \text{in}\ s\ \{\Psi\}}\ \text{T-NewX}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1, e_2, y\ \text{ok} \quad L = \bigcup_{0 \le i < e_1} \{e_2 + |X|_\Gamma * i\} \quad range_\Gamma\ X = \tau \quad y, X, Un, \tau \in \Delta \\ \sigma_1 = copy_\Gamma\ L\ from\ H\ to\ X \quad \Phi, \sigma_2 = chkAndRem_\Gamma\ \tau\ L \quad \sigma_3 = updUn_\Gamma\ L\ \tau\ \bot\end{array}}{\Gamma;\Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := \text{bless}_X[e_1]\ e_2\ \{\Psi\}}\ \text{T-Bless}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1, e_2, y\ \text{ok} \quad L = \bigcup_{0 \le i < e_1} \{e_2 + |X|_\Gamma * i\} \quad range_\Gamma\ (X) = \tau \quad y, X, Un, \tau \in \Delta \\ \sigma_1 = copy_\Gamma\ L\ from\ H\ to\ \tau \quad \Phi, \sigma_2 = chkAndRem_\Gamma\ X\ L \quad \sigma_3 = updUn_\Gamma\ L\ \tau\ 1\end{array}}{\Gamma;\Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := \text{unbless}_X\ [e_1]\ e_2\ \{\Psi\}}\ \text{T-UnBless}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1, e_2\ \text{ok} \quad X, H \in \Delta \quad X \neq Un \quad v_h = readFrom_\Gamma\ H\ (e_1{:}X) \quad v_x = X_\Gamma(e_1) \\ H_1 = H[(e_1 + offset_\Gamma\ X\ p) \leftarrow e_2] \quad \sigma_1 = copy_\Gamma\ e_1\ from\ H_1\ to\ X \quad \sigma = \sigma_1 \circ [H_1/H]\end{array}}{\Gamma;\Delta \vdash \{e_1 \in dom_\Gamma\ X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ X(e_1).p := e_2\ \{\Psi\}}\ \text{T-Write}$$

$$\frac{\begin{array}{ccc}\Gamma \vdash e, y\ \text{ok} & y \in \Delta & X \neq Un \\ v_h = readFrom_\Gamma\ H\ (e{:}X) & v_x = X_\Gamma(e) & \sigma = [(H_1(e + offset_\Gamma\ X\ p))/y]\end{array}}{\Gamma;\Delta \vdash \{e \in dom_\Gamma\ X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ y := X(e).p\ \{\Psi\}}\ \text{T-Read}$$

$$\frac{\Gamma \vdash e\ \text{ok} \quad v_h = readFrom_\Gamma\ H\ (e{:}X) \quad v_x = X_\Gamma(e) \quad \Gamma;\Delta \vdash \{\Phi_1\}\ s_1\ \{\Psi\} \quad \Gamma;\Delta \vdash \{\Phi_2\}\ s_2\ \{\Psi\}}{\Gamma;\Delta \vdash \{((e \in dom_\Gamma\ X \wedge (X = Un \vee v_h = v_x)) \Rightarrow \Phi_1) \wedge (e \notin dom_\Gamma\ X \Rightarrow \Phi_2)\}\ \text{if}\ e\ \text{is in}\ X\ \text{then}\ s_1\ \text{else}\ s_2\ \{\Psi\}}\ \text{T-IsX}$$

$$\frac{\Gamma \vdash e_1, e_2\ \text{ok} \quad H_1 = H[e_1 \leftarrow e_2] \quad \sigma = [H_1/H]}{\Gamma;H \vdash \{\sigma(\Psi)\}\ \text{lib}\ e_1 := e_2\ \{\Psi\}}\ \text{T-LibWrite} \qquad \frac{\Gamma \vdash e, y\ \text{ok} \quad \sigma = [(He)/y]}{\Gamma;y \vdash \{\sigma(\Psi)\}\ \text{lib}\ y := e\ \{\Psi\}}\ \text{T-LibRead}$$

$$\frac{\Gamma \vdash e_1\ \text{ok} \quad \Gamma;\Delta \vdash \{\Phi_1\}\ s_1\ \{\Psi\} \quad \Gamma;\Delta \vdash \{\Phi_2\}\ s_2\ \{\Psi\}}{\Gamma;\Delta \vdash \{(e_1 = 0 \Rightarrow \Phi_1) \wedge (e_1 \neq 0 \Rightarrow \Phi_2)\}\ \text{if}\ e_1\ \text{then}\ s_1\ \text{else}\ s_2\ \{\Psi\}}\ \text{T-If}$$

$$\frac{\Gamma \vdash e\ \text{ok} \quad \Gamma \vdash \Psi_{inv}\ \text{ok} \quad \Gamma;\Delta \vdash \{\Phi\}\ s\ \{\Psi_{inv}\}}{\Gamma;\Delta \vdash \{\Psi_{inv} \wedge (e_1 \neq 0 \Rightarrow \Phi) \wedge (e_1 = 0 \wedge \Psi_{inv} \Rightarrow \Psi)\}\ \text{while}\ e\ s\ \{\Psi\}}\ \text{T-While}$$

$$\frac{\Gamma;\Delta \vdash \{\Phi_1\}\ s_2\ \{\Psi\} \quad \Gamma;\Delta \vdash \{\Phi\}\ s_1\ \{\Phi_1\}}{\Gamma;\Delta \vdash \{\Phi\}\ s_1; s_2\ \{\Psi\}}\ \text{T-Seq} \qquad \frac{}{\Gamma;\Delta \vdash \{\Psi\}\ \text{skip}\ \{\Psi\}}\ \text{T-Skip} \qquad \frac{}{\Gamma;\Delta \vdash \{\mathit{True}\}\ \text{abort}\ \{\Psi\}}\ \text{T-Abort}$$

Figure 14. A Floyd-Hoare logic for YCORE

$\boxed{\Gamma \vdash a : t}$ well-typed terms, where $t ::= \tau \mid \hat{\tau} \mid \textit{int set}$

$$\frac{}{\Gamma \vdash i : \textit{int}} \qquad \frac{x \in \textit{dom}\ \Gamma}{\Gamma \vdash x : \textit{int}} \qquad \frac{\Gamma \vdash a_1 : \textit{int} \quad \Gamma \vdash a_2 : \textit{int}}{\Gamma \vdash a_1\ \textit{op}\ a_2 : \textit{int}} \qquad \frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : (\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash a : (\tau_1, \tau_2)}{\Gamma \vdash a.i : \tau_i}$$

$$\frac{\Gamma(X) = \hat{\tau}}{\Gamma \vdash X : \hat{\tau}} \qquad \frac{}{\Gamma \vdash \bot : \tau} \qquad \frac{\Gamma, \ell \vdash \hat{e} : \tau}{\Gamma \vdash \lambda \ell.\hat{e} : \textit{int} \to \tau} \qquad \frac{\Gamma \vdash a : \textit{int} \quad \Gamma \vdash a' : \textit{int set} \quad \Gamma \vdash \hat{e}_1 : \tau \quad \Gamma \vdash \hat{e}_2 : \tau}{\Gamma \vdash \text{if } a \in a' \text{ then } \hat{e}_1 \text{ else } \hat{e}_2 : \tau}$$

$$\frac{\Gamma \vdash \hat{v} : \textit{int} \to \tau \quad \Gamma \vdash v : \textit{int}}{\Gamma \vdash \hat{v}\ v : \tau} \qquad \frac{\Gamma \vdash a : \hat{\tau}}{\Gamma \vdash \textit{dom}\ a : \textit{int set}} \qquad \frac{\Gamma, x \vdash \Phi\ \text{ok}}{\Gamma \vdash \{x \mid \Phi\} : \textit{int set}} \qquad \frac{\Gamma \vdash (a.i).p : \tau \quad p \neq \cdot}{\Gamma \vdash a.ip : \tau}$$

$\boxed{\Gamma \vdash \vec{e}\ \text{ok}}$ generalizing well-formedness to lists of expressions

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash \vec{e}\ \text{ok}}{\Gamma \vdash e_1, \vec{e}\ \text{ok}} \qquad \frac{}{\Gamma \vdash \cdot\ \text{ok}}$$

$\boxed{\Gamma \vdash \tau\ \text{ok}}$ well-formed types

$$\frac{}{\Gamma \vdash \textit{int}\ \text{ok}} \qquad \frac{X \in \textit{dom}\ \Gamma}{\Gamma \vdash X\ \text{ok}} \qquad \frac{\Gamma \vdash \tau_1\ \text{ok} \quad \Gamma \vdash \tau_2\ \text{ok}}{\Gamma \vdash (\tau_1, \tau_2)\ \text{ok}}$$

$\boxed{\Gamma \vdash \Phi\ \text{ok}}$ well-formed formulas

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash a' : t}{\Gamma \vdash a = a'\ \text{ok}} \qquad \frac{\Gamma \vdash a : \textit{int} \quad \Gamma \vdash a' : \textit{int set}}{\Gamma \vdash a \in a'\ \text{ok}} \qquad \frac{\Gamma \vdash a : \textit{int} \quad \Gamma \vdash a' : \textit{int}}{\Gamma \vdash a < a'\ \text{ok}}$$

$$\frac{\Gamma \vdash \Phi\ \text{ok}}{\Gamma \vdash \neg\Phi\ \text{ok}} \qquad \frac{\Gamma \vdash \Phi\ \text{ok} \quad \Gamma \vdash \Psi\ \text{ok}}{\Gamma \vdash \Phi \wedge \Psi\ \text{ok}} \qquad \frac{\Gamma \vdash \Phi\ \text{ok} \quad \Gamma \vdash \Psi\ \text{ok}}{\Gamma \vdash \Phi \vee \Psi\ \text{ok}}$$

$$\frac{\Gamma, x \vdash \Phi\ \text{ok}}{\Gamma \vdash \forall x.\Phi\ \text{ok}} \qquad \frac{\Gamma, X{:}\hat{\tau} \vdash \Phi\ \text{ok}}{\Gamma \vdash \forall X{:}\hat{\tau}.\Phi\ \text{ok}} \qquad \frac{\Gamma, x \vdash \Phi\ \text{ok}}{\Gamma \vdash \exists x.\Phi\ \text{ok}} \qquad \frac{\Gamma, X{:}\hat{\tau} \vdash \Phi\ \text{ok}}{\Gamma \vdash \exists X{:}\hat{\tau}.\Phi\ \text{ok}}$$

$\boxed{\vdash E : \Gamma}$ store typing

$$\frac{\vdash E : (\Gamma_2, \Gamma_1)}{\vdash E : (\Gamma_1, \Gamma_2)} \qquad \frac{\vdash v : \textit{int} \quad \vdash E : \Gamma}{\vdash E, (x \mapsto v) : \Gamma, x} \qquad \frac{\vdash \hat{v} : \hat{\tau} \quad \vdash E : \Gamma}{\vdash E, (X \mapsto (\hat{v}{:}\hat{\tau})) : \Gamma, X{:}\hat{\tau}}$$

$$\frac{\vdash \hat{v} : \textit{int} \to \textit{int} \quad \vdash E : \Gamma}{\vdash E, (\mathit{Un} \mapsto (\hat{v}{:}\textit{int} \to \textit{int})) : \Gamma, \mathit{Un}{:}\textit{int} \to \textit{int}} \qquad \frac{\vdash \hat{v} : \textit{int} \to \textit{int} \quad \models \forall \ell.\ell \in \textit{dom}\ \hat{v}}{\vdash (H \mapsto (\hat{v}{:}\textit{int} \to \textit{int})) : (H{:}\textit{int} \to \textit{int})}$$

$\boxed{\vdash \Gamma\ \text{ok}}$

$$\frac{}{\vdash \cdot\ \text{ok}} \qquad \frac{\Gamma\ \text{ok} \quad x \notin \textit{dom}\ \Gamma}{\vdash \Gamma, x\ \text{ok}} \qquad \frac{\Gamma\ \text{ok} \quad X \notin \textit{dom}\ \Gamma \quad \Gamma \vdash \tau\ \text{ok}}{\vdash \Gamma, X{:}\tau\ \text{ok}}$$

$\boxed{\vdash \Gamma; \Delta\ \text{ok}}$

$$\frac{\vdash \Gamma\ \text{ok}}{\vdash \Gamma; \cdot\ \text{ok}} \qquad \frac{\vdash \Gamma; \Delta\ \text{ok} \quad x \in \textit{dom}\ \Gamma \quad x \notin \Delta}{\vdash \Gamma; \Delta, x\ \text{ok}}$$

Figure 15. Well-formed terms, formulas, stores, and environments

$\boxed{[\![a]\!]_E}$ interpretation of terms

$$
\begin{array}{lcl}
[\![x]\!]_E & = & E(x) \\
[\![X]\!]_E & = & E(X) \\
[\![v]\!]_E & = & v \\
[\![\hat{v}]\!]_E & = & \hat{v} \\
[\![a.0]\!]_E & = & v_1 \text{ when } [\![a]\!]_E = (v_1, v_2) \\
[\![a.1]\!]_E & = & v_2 \text{ when } [\![a]\!]_E = (v_1, v_2) \\
[\![a.ip]\!]_E & = & [\![[\![a.i]\!]_E.p]\!]_E \text{ when } p \neq \cdot \\
[\![\lambda x.\hat{e}\ \ell]\!]_E & = & [\![\hat{e}[\ell/x]]\!]_E \\
[\![\text{if } a \in a' \text{ then } \hat{e} \text{ else } \hat{e}']\!]_E & = & [\![\hat{e}]\!]_E \text{ when } [\![a]\!]_E \in [\![a']\!]_E \\
[\![\text{if } a \in a' \text{ then } \hat{e} \text{ else } \hat{e}']\!]_E & = & [\![\hat{e}']\!]_E \text{ when } [\![a]\!]_E \notin [\![a']\!]_E \\
[\![\{x \mid \Phi\}]\!]_E & = & \{v \mid E \models \Phi[v/x]\} \\
[\![dom\ a]\!]_E & = & [\![\ell \mid a\ \ell \neq \bot]\!]_E
\end{array}
$$

$\boxed{E \models \hat{v}_1 \cong \hat{v}_2}$ extensional equality on map values

$$\frac{\forall l. E \models (\hat{v}_1 l) = (\hat{v}_2 l)}{E \models \hat{v}_1 \cong \hat{v}_2}$$

$\boxed{E \models \Phi}$ interpretation of formulas

$$
\begin{array}{lcll}
E \models \mathit{True} & & & \\
E \models \neg\Phi & \iff & E \models \Phi \text{ is invalid} & \\
E \models \Phi_1 \wedge \Phi_2 & \iff & E \models \Phi_1 \text{ and } E \models \Phi_2 & \\
E \models \Phi_1 \vee \Phi_2 & \iff & E \models \Phi_1 \text{ or } E \models \Phi_2 & \\
E \models \forall x.\Phi & \iff & \text{for all integers } i, E \models \Phi[i/x] & \\
E \models \forall X{:}\hat{\tau}.\Phi & \iff & \text{for all map values } \hat{v}{:}\hat{\tau}, E \models \Phi[\hat{v}/X] & \\
E \models \exists x.\Phi & \iff & \text{for all some integer } i, E \models \Phi[i/x] & \\
E \models \exists X{:}\hat{\tau}.\Phi & \iff & \text{for some map value } \hat{v}{:}\hat{\tau}, E \models \Phi[\hat{v}/X] & \\
E \models a_1 = a_2 & \iff & [\![a_1]\!]_E = [\![a_2]\!]_E & \text{when } \vdash E : \Gamma \text{ and } \Gamma \vdash a_i : \tau \\
E \models a_1 = a_2 & \iff & E \models [\![a_1]\!]_E \cong [\![a_2]\!]_E & \text{when } \vdash E : \Gamma \text{ and } \Gamma \vdash a_i : \hat{\tau} \\
E \models a_1 \in a_2 & \iff & [\![a_1]\!]_E \in [\![a_2]\!]_E & \\
E \models a_1 < a_2 & \iff & [\![a_1]\!]_E < [\![a_2]\!]_E &
\end{array}
$$

$\boxed{\Gamma \models \Phi}$ $\Gamma \models \Phi \iff$ for all E such that $\vdash E : \Gamma$, we have $E \models \Phi$

Figure 16. Interpretation of terms and formulas