Modular Protections against Non-control Data Attacks

Cole Schlesinger[†] Karthik Pattabiraman[‡] Nikhil Swamy[*] David Walker[†] Benjamin Zorn[*]
[†]*Princeton University*
{*cschlesi,dpw*}*@cs.princeton.edu*
[‡]*University of British Columbia*
karthikp@ece.ubc.ca
[*]*Microsoft Research*
{*nswamy,zorn*}*@microsoft.com*

***Abstract*—This paper introduces YARRA, a conservative extension to C to protect applications from non-control data attacks. 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 evaluate a prototype implementation of a compiler and runtime system for YARRA by using it to harden four common server applications against known non-control data vulnerabilities. We show that YARRA defends against these attacks with only a negligible impact on their end-to-end performance.**

***Keywords*-language-based security; non-control data attack; data integrity; control-flow integrity; Hoare logic; frame rule; data isolation**

I. INTRODUCTION

Most important applications contain components written in unsafe languages such as C and C++. These components are vulnerable to a variety of memory corruption attacks. To develop comprehensive protections for these unsafe components, it is essential to identify wide, prominent classes of attacks, to analyze such classes mathematically, and to implement and evaluate effective solutions against them.

One broad class of attack on unsafe programs is the *control-based attack*, in which an attacker uses a memory corruption error, such as a buffer overflow or use-after-free, to overwrite control-data such as a return address or function pointer and thereby modifies the control-flow of the program. Through the early to mid 2000s, both industry and academia developed mitigation techniques against control-data attacks. One particularly noteworthy piece of work in this line of inquiry, due to Abadi *et al.* [1], developed a formal model of *control-flow integrity* and used this model to prove the correctness of defenses against a formal attacker.

In this paper, we analyze a separate class of attacks: *non-control data attacks*. 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. In 2005, Chen *et al.* [9] demonstrated that such non-control data attacks are a serious threat against many real applications, including widely-used server programs. Since then, due to the mitigations that have been developed against control-based attacks, the prevalence of non-control data attacks has increased [29].

Non-control data attacks are orthogonal to control-based attacks in the sense that provably secure control-flow attack defenses (such as Abadi's) may be completely vulnerable to non-control data attacks. Conversely, provably secure defenses against non-control data attacks may be vulnerable to control-flow attacks. Defense against both attack classes may be implemented through a union of orthogonal solutions. Hence, in this paper, we set aside the problem of control-based attacks to focus squarely on non-control data attacks. The following paragraphs summarize our key contributions.

A modular solution to non-control data attacks. Our solution takes the form of a language extension to C, which we call YARRA. 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*, and YARRA ensures that data with such types are impervious to non-control data attacks.

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 static critical type or *not to use* a pointer with a static critical type. When accessing data through a pointer with a static critical type, a programmer declares that she expects the underlying memory to have that same critical type dynamically. When reading or writing through a pointer that, statically, does not have a critical type, the programmer declares that she does not expect to be accessing memory with dynamic critical type.

This design has a number of advantages. First, it is simple to understand and easy to use. Every programmer is familiar with the concept that the underlying dynamic type of a data structure should match the static type of the pointer. YARRA merely puts an enforcement mechanism for this concept in place. Violation of this property, and the subsequent unintended modification of a critical data type, is at the heart of all non-control data attacks.

Second, our design supports adaptation of legacy code with minimal effort: type declarations may be added to an existing code base, literally one at a time, incrementally hardening a program against non-control data attacks.

Third, the design is highly modular in the sense that once a module is proved secure, it may be linked with arbitrary, unverified library code, and that library will be unable to wage a non-control data attack against it. In contrast, systems

such as Cyclone [12], CCured [23], Softbound [21] 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, like control-flow integrity, remains a very useful technique).

***Formal safety and modularity properties for* YARRA.** 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 key element of our logic is a new kind of *type-based frame rule*. This frame rule allows components responsible for implementing security infrastructure to be verified independently of the *unverified*, possibly buggy and vulnerable libraries that they are linked with. Despite such bugs and vulnerabilities, these libraries cannot wage non-control data attacks against the verified security components. Consequently, the frame rule codifies the modularity properties that YARRA programmers may rely upon. The proof of soundness of our program logic, including this novel frame rule, is the deep theoretical result of our work.

A formal definition of non-control data attacks. Inherent in our safety proof, and our analysis of the frame rule, is a formal, language-based definition of non-control data attacks. To be specific, a non-control data attack is any attack driven by a sequential, imperative program with fixed, static control-flow and the license to attempt unlimited reads and writes (including writes outside the normal bounds of data structures such programs allocate). The attacks are waged against YARRA programs, which are also defined to have fixed, static control flow. We limit the control constructs in our formal model because that is the simplest, clearest way to define the essence of a non-control data attack (as opposed to a control-based attack) and thereby to characterize the problem and our solution. We leave an analysis of multi-threaded programs to future work.

***Implementation of* YARRA.** The semantics of YARRA may be implemented in more than one way. Different implementations have different performance trade-offs in terms of time and space and different requirements in terms of access to source code for transformation. We have implemented a compiler and run-time system for YARRA that supports two different runtime enforcement modes. The first mode, inspired by previous work on Write Integrity Testing (WIT) [2], instruments source code with dynamic checks that cannot be proven unnecessary at compile time. The second mode, inspired by previous work on Samurai [24], 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.

Experimental evaluation. We demonstrate the effectiveness of YARRA on a collection of important server applications including SSH, telnet, HTTP and FTP with security-sensitive data that may be vulnerable to non-control data attacks. These applications typically contain, amongst thousands of lines of code, a relatively small, clearly defined module, or set of modules that implement important security considerations and require careful auditing—applications with this structure are best suited to the protections that YARRA can provide. For these applications, we observe that our implementation has negligible overhead relative to the end-to-end performance of the application as a whole. In addition, the programmer integration effort was on the order of a few hundred modified lines of code or less in applications tens of thousands of lines long.

For a more thorough, but artificial, measurement of the performance impact of YARRA, we adapt BGET [32], a widely used memory manager, to use YARRA to protect the allocator's internal data structures from corruptions by the application. When used in such a scenario, where a large number of data accesses involve critical data types, we find that the performance overhead can be very substantial.

One conclusion we draw from these experiments is that our current prototype, though completely unoptimized, is an eminently practical defense against non-control data attacks in typical server applications where the amount of critical data that needs to be protected is relatively small.

II. 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 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 vulnerabilities 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:

- *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 [21], 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.

A. *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 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 Log 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 a 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 III in terms of a type-based *frame rule* and prove it sound.

B. 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[1]. Likewise, it is an error to attempt to unbless memory from type T when that memory location had not previously been blessed at 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.

[1]An illegal cast of this sort might invalidate 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");
29 }
```

Figure 1. A simplified memory manager

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 [32], 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.

III. Semantics of Yarra

This section defines YCORE, a sequential, imperative language intended to serve as a core model for YARRA. This formal development serves two purposes. First, YCORE's semantics makes precise our attacker model: the attacker is represented by calls to unverified library code that may have arbitrary effects on the heap, but cannot alter the control flow of the program. Second, we define robustness in the presence of non-control data attacks to be the ability to reason locally about critical data structures, even in the presence of arbitrary heap effects caused by library code.

We formulate robustness, or modular local reasoning, in the context of a program logic for YCORE programs and we show that this logic admits a frame rule. Unlike recent presentations of the frame rule that require the use of separation logic [26], ours is in the context of a classical Hoare logic and relies on the type structure of the program for modular reasoning. In addition to its technical novelty, we argue that our type-based approach provides a more familiar model for programmers already used to working with types. Furthermore, unlike in other logics, YARRA's dynamic protections make our frame rule sound even in the presence of heap effects caused by unverified libraries. As such, this frame rule captures the essence of YARRA's modular protections against non-control data attacks.

A. Syntax

Broadly speaking, YCORE is a simple *while*-language, augmented with critical type declarations, and memory operations to manipulate critical memory. Figure 2 shows the syntax of YCORE, starting with our meta variable conventions. Integer constants are i, j, ℓ, where, we generally use ℓ for memory locations. Local variables are x, y, z, and critical data types (and their representations as maps) are X, Y, Z with H and Un being two distinguished map names.

Expressions e are purely arithmetic terms, built from integer constants, integer variables and primitive operators op. Values v are either integer constants i, or are structured tuples (v_1, v_2) corresponding to the values of protected object

integer const.			i, j, ℓ
local variables			x, y, z
map names			X, Y, Z, H, Un
values	v	::=	$i \mid (v_1, v_2)$
expr.	e	::=	$i \mid x \mid e\ op\ e'$
stmt./hole	s	::=	skip | if e then s_1 else s_2 | while e s
sequence			$\mid s_1; s_2$
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_{base}
unbless e objs. starting at e_{base}			$\mid y :=$ unbless$_X$ $[e]$ e_{base}
dynamic typecase			$\mid$ if e is in X then s_1 else s_2
checked read			$\mid y := X(e).p$
un-checked read			$\mid$ lib $y := e$
checked write			$\mid X(e_1).p := e_2$
un-checked write			$\mid$ lib $e_1 := e_2$
dynamic failure			$\mid$ abort
hole			$\mid \bullet_i$
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 $\hat{e}'$
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	::=	$\cdot \mid \sigma, [a/X] \mid \sigma, [a/x]$
mod. set	Δ	::=	$\cdot \mid \Delta, X \mid \Delta, x$
static env.	Γ	::=	$\cdot \mid \Gamma, X{:}\hat{\tau} \mid \Gamma, x$
runtime env.	E	::=	$E, x \mapsto i \mid E, X \mapsto (\hat{v}{:}\hat{\tau})$
		$\mid$	$H \mapsto (\hat{v}{:}\hat{\tau}), Un \mapsto (\hat{v}{:}\hat{\tau})$
either env.	$\mathcal{E}$	::=	$\Gamma \mid E$

Figure 2. Syntax of YCORE

types. Note, expressions do not include tuples, ensuring that well-scoped expressions always evaluate to integers.

Basic statements include the usual forms for branching, looping, sequencing, assertions, and 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 statement form s also serves as a multi-hole context, where the holes $\bullet_1, \ldots, \bullet_n$ represent points at which control transfers to an attacker program. We write $s[s_i]_i$ to replace hole i in s with s_i. We write $s[s_1, \ldots, s_n]$ for the hole-free statement obtained by replacing each hole $\bullet_i$ in s with s_i. We place specific conditions on the attacker code that can be used to fill a hole in Section III-D.

Critical type commands. The statement 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 II. Here, the command ($y :=$ bless$_X[e]$ e_{base}) operates on an *array of locations* starting at the location e_{base} and including e objects each to be protected at the type X (where e is expected to evaluate to a non-negative integer). The returned value y is a reference to the start of the array of newly blessed objects. Analogously, the command ($y :=$ unbless$_X$ $[e]$ e_{base}) removes

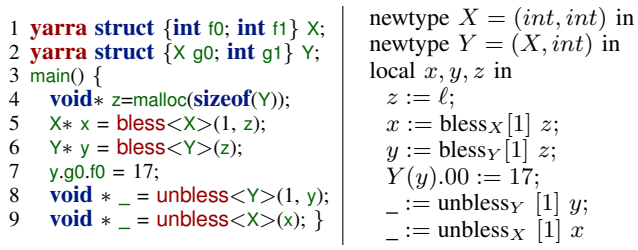

Figure 3. Relating the syntax of YARRA to YCORE

protections on an array of critical objects. The dynamic typecase (if e is in X then s_1 else s_2) statement is useful for modeling the vacant command of Section II-B, as well as other constructs—it can be used to check whether a location e holds a critical object of type X.

Reading and writing memory. YCORE includes two forms each of read and write instructions. A checked read ($y := X(e).p$) 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 un-checked read instruction (lib $y := e$) reads the contents of an arbitrary memory location e from the heap H into a local variable y. Similarly, a checked write ($X(e_1).p := e_2$) attempts to write to a structured type using a field assignment; un-checked writes (lib $e_1 := e_2$) modify a single location e_1 in the heap, overwriting its contents with e_2. We use the un-checked forms to model the actions of arbitrary, untrusted code, *e.g.*, third party libraries.

Failure modes. We model two 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 stuck programs.

Types and the assertion language. The type language of YCORE includes int, pairs, and 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 nested pairs. We omit unions. 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. Maps are lambda-terms ($\lambda\ell.\hat{e}$), with types described using the map types $\hat{\tau}$. The body ($\hat{e}$) of a map value is built from values v, an application form, a conditional form, and a distinguished value $\perp$ used to model partial maps.

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.(This is not an unusual choice in systems governed by classical logics. See, for example, work on Havoc [15].) 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. In cases (*e.g.*, lines 6 and 9) where we omit the first argument to bless or unbless, the argument defaults to 1.

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.

B. Dynamic semantics

Figure 4 shows selected rules from the dynamic semantics of YCORE—our technical report [28] includes the full definition. The semantics is a small-step reduction relation of the form $(E; s) \rightsquigarrow (E'; s')$, where (E, s) is called a *run-time configuration*. Such configurations contain *run-time environments* E and hole-free statements s.

Runtime environments E contain integer assignments for local variables ($x \mapsto i$); a typed map value ($\hat{v}{:}\hat{\tau}$) for each critical type X defined in the program ($X \mapsto \hat{v}{:}\hat{\tau}$); a map value for the conventional heap ($H \mapsto \hat{v}{:}\hat{\tau}$); and, finally, a map value for *Un*, the collection of unblessed locations ($Un \mapsto \hat{v}{:}\hat{\tau}$). We call each map value $\hat{v}$ in E a *heaplet*. The heaplet for a critical type 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, *i.e.*, in a well-formed environment containing $X \mapsto (\hat{v}{:}\hat{\tau})$, $\hat{v}$ is a partial map of type $\hat{\tau}$, where $\hat{\tau} = int \rightarrow X$. The heap H is a total map from memory addresses to integers (*i.e.*, it has type $int \rightarrow int$), while *Un* is a partial map of type $int \rightarrow int$. The totality of the H-map is simply a technical convenience—we could, with a little additional book-keeping, allow H to be a partial map.

Auxiliary functions. Figure 5 defines several auxiliary functions used throughout the semantics. These functions are straightforward, although a few comments are worthwhile. First, note that most of our auxiliary functions carry indexes (subscripted) that represents environment arguments. For example, $[\![e]\!]_E$ is a standard denotational semantics for

$$\frac{\hat{\tau} = \mathit{int} \to \tau}{(E; \text{newtype } X = \tau \text{ in } s) \rightsquigarrow (E, X \mapsto (\lambda\ell.\bot{:}\hat{\tau}); s)}\text{E-NewX} \qquad \frac{[\![e_1]\!]_E = \ell \quad E' = E[y \mapsto H_E(\ell)]}{(E; \text{lib } y := e_1) \rightsquigarrow (E'; \text{skip})}\text{E-LibRd}$$

$$\frac{[\![e_1]\!]_E = \ell \quad [\![e_2]\!]_E = v \quad E(H) = \hat{v}{:}\hat{\tau} \quad E' = E[H \mapsto (\hat{v}[\ell \leftarrow v]{:}\hat{\tau})]}{(E; \text{lib } e_1 := e_2) \rightsquigarrow (E'; \text{skip})}\text{E-LibWr} \qquad \frac{[\![e]\!]_E = \ell \quad \ell \in \mathit{dom}_E X \quad X_E(\ell) \neq \mathit{readFrom}_E\ H\ (\ell{:}X)}{(E; y := X(e).p) \rightsquigarrow (E; \text{abort})}\text{E-RdAbort}$$

$$\frac{p \neq \cdot \quad [\![e_1]\!]_E = \ell \quad \ell \in \mathit{dom}_E X \quad X_E(\ell) = \mathit{readFrom}_E\ H\ (\ell{:}X) \quad \ell' = \ell + \mathit{offset}_E\ X\ p \quad E' = E[y \mapsto H_E(\ell')]}{(E; y := X(e_1).p) \rightsquigarrow (E'; \text{skip})}\text{E-Rd}$$

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

Figure 4. $(E; s) \rightsquigarrow (E'; s')$: Dynamic semantics of YCORE (Selected rules)

$[\![e]\!]_E$ standard denotation of expressions (see TR)

$$\begin{array}{lcl}
\mathit{dom}_E X & = & \{\ell \mid X_{\mathcal{E}}(\ell) \neq \bot\} \\
\mathit{dom}_\Gamma X & = & \mathit{dom}\ X \\
\mathit{range}_E\ X & = & \tau \ \textbf{ when } E(X) = (\hat{v}{:}\mathit{int} \to \tau) \\
\mathit{range}_\Gamma\ X & = & \tau \ \textbf{ when } \Gamma(X) = \mathit{int} \to \tau \\
X_E(\ell) & = & [\![\hat{v}\ \ell]\!]_E \ \textbf{ when } E(X) = (\hat{v}{:}\hat{\tau}) \\
X_\Gamma(\ell) & = & X\ \ell \\
a_m[a \leftarrow a'] & = & \lambda\ell.\text{if } \ell \in \{a\} \text{ then } a' \text{ else } (a_m\ \ell) \\
|\mathit{int}|_{\mathcal{E}} & = & 1 \\
|Y|_{\mathcal{E}} & = & |\mathit{range}_{\mathcal{E}}\ Y|_{\mathcal{E}} \\
|(\tau_1, \tau_2)|_{\mathcal{E}} & = & |\tau_1|_{\mathcal{E}} + |\tau_2|_{\mathcal{E}} \\
\mathit{offset}_{\mathcal{E}}\ \mathit{int}\ \cdot & = & 0 \\
\mathit{offset}_{\mathcal{E}}\ (\tau_1, \tau_2)\ 0p & = & \mathit{offset}_{\mathcal{E}}\ \tau_1\ p \\
\mathit{offset}_{\mathcal{E}}\ (\tau_1, \tau_2)\ 1p & = & |\tau_1|_{\mathcal{E}} + \mathit{offset}_{\mathcal{E}}\ \tau_2\ p \\
\mathit{offset}_{\mathcal{E}}\ Y\ p & = & \mathit{offset}_{\mathcal{E}}\ (\mathit{range}_{\mathcal{E}}\ Y)\ p \\
\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}\mathit{int}) & = & Y_{\mathcal{E}}(\ell) \\
\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}Z) & = & \mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}(\mathit{range}_{\mathcal{E}}\ Z)) \\
\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}(\tau_1, \tau_2)) & = & (v_1, v_2) \\
\quad \text{where } v_1 = \mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}\tau_1) & & \\
\quad \text{and } \ v_2 = \mathit{readFrom}_{\mathcal{E}}\ Y\ ((\ell + |\tau_1|_{\mathcal{E}}){:}\tau_2) & &
\end{array}$$

Figure 5. Auxiliary functions

expressions, defined relative to the assignments of local variables in E. Some function symbols are indexed either by runtime environments E or static environments Γ. This allows us to overload function symbols for use in both the static and dynamic semantics. For example, $\mathit{dom}_E X$, used in the dynamic semantics, concretely represents the domain of a map X as the set of locations on which X does not evaluate to $\bot$. Statically, $\mathit{dom}_\Gamma(X)$ is simply a term $\mathit{dom}\ X$ in the logic. Many of the functions in Figure 5 are parametric in their environment index—these functions carry the index $\mathcal{E}$, where $\mathcal{E}$ may be either E or Γ.

A brief description of each of the auxiliary functions follows: $X_{\mathcal{E}}(\ell)$ is the value of the map X at the location ℓ; $a_m[a \leftarrow a']$ updates the map a_m at location a to contain a'; $\mathit{range}_{\mathcal{E}}\ X$ is the range type of a map; $|\tau|_{\mathcal{E}}$ represents the size (in machine words) of a value v of type τ; $\mathit{offset}_{\mathcal{E}}\ \tau\ p$ is the offset of a field accessed via the path p in the type τ; $\mathit{readFrom}_{\mathcal{E}}\ X\ (\ell{:}\tau)$ reads a structured value at location ℓ of type τ from the map X. Note that $\mathit{offset}_\Gamma\ \tau\ p$ is a partial function, *e.g.*, $\mathit{offset}_{\mathcal{E}}\ ((\mathit{int}, \mathit{int}), \mathit{int})\ 0$ is undefined. This ensures that only word-length int-valued fields in a nested tuple type can be directly addressed. Second, $\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}\tau)$ 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.

We turn now to a discussion of the rules in Figure 4.

Heaplets for new critical types. The rule (E-NewX) shows the initialization of an empty heaplet (everywhere $\bot$) for a new critical type X. Structured values corresponding to the objects of the critical type X are added to the X heaplet whenever the program issues a bless command; values are removed from the heaplet when unblessed. As such, the heaplet X serves as a backing store for X values. For space reasons, we do not show the dynamic rules for blessing and unblessing—there are several subtleties related to blessing and unblessing nested objects. However, we present the axiomatic semantics of these commands in Section III-C.

Un-checked reads and writes. The rule (E-LibRd) shows the reduction of a read operation performed by untrusted code. We evaluate the pure expression e to a location ℓ, and update the local variable y in the environment to hold the value in the heap H at location ℓ. (E-LibWr) is also unsurprising—we simply update the heap H at the location ℓ to the value v. The important aspect of these rules is that library reads and writes only have effect on the heap H and on local variables in scope, but never update the heaplets for any critical type X. It is possible to implement this semantics for un-checked writes in multiple ways. For example, in its library protection mode, our compiler uses hardware page protections to maintain the integrity of critical heaplets.

Checked reads. Although library instructions cannot modify the critical heaplets, errant writes by a library can corrupt a critical object stored in the heap. We use the backing store provided by the critical heaplets to detect such corruptions and abort the program, if necessary. The rules (E-RdAbort) and (E-Rd) show this behavior. When reducing $y := X(e).p$ we evaluate e to a location ℓ and check that ℓ is a reference to a blessed object. A failure of this first check causes the

configuration to get stuck, a situation prevented by the static semantics. Next, we check that the value in the backing store X at location ℓ matches the value stored in the heap at the same location. If this check fails, the program aborts. Otherwise, we compute the offset of the field being read, and update the local y with the contents of the field.

Note that as shown here, since the critical heaplet for X always holds an uncorrupted value, we might recover from a corruption instead of aborting. However, we aim to provide an abstract semantics for YCORE that is independent of the specific choice of implementing critical heaplets. In particular, rather than storing copies of objects in the critical heaplets, we may wish to use our compiler's source protection mode, or to resort to other forms of protections that, say, only maintain checksums or cryptographic digests rather than full shadow copies. Such implementation strategies allow memory corruption to be detected, but may not support recovery. By allowing (E-RdAbort) to fail when a corruption is detected, we provide YARRA with the flexibility to choose among various implementation strategies.

Checked writes. (E-Wr) shows the reduction of an instruction that writes via an X-typed reference. As for checked reads, we ensure that the location being written to is in the domain of the X heaplet (otherwise the configuration is stuck) and check, using the backing store, that the critical object being modified is uncorrupted (and abort otherwise, using (E-WrtAbort) a rule analogous to (E-RdAbort)). We then update H at the appropriate location and offset, and, importantly, in the last premise, we copy the updated object from the heap into the critical heaplet X. Thus, abstractly, writes through typed references correspond to a pair of writes, both to the heap and to the critical object's shadow copy. However, the YARRA implementation may or may not actually manifest the update to the shadow copy, *e.g.*, when using our source protection mode.

Intuitively, one can imagine that YCORE programs enjoy a measure of data integrity, since copies of critical objects are maintained in uncorruptible backing stores. The next section makes this notion of data integrity precise. Specifically, we show that despite the presence of arbitrary heap modification by untrusted code, programmers can reason about the invariants of critical objects using modular, local reasoning principles. The crux of this idea is embodied by the frame rule in a program logic for YCORE, presented next.

C. Static semantics

The static semantics of YCORE is given by the relation $\Gamma; \Delta \vdash \{\Phi\}\ s\ \{\Psi\}$, a classical Floyd-Hoare logic judgment. The judgment 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 context Γ contains a 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 (like runtime environments E) 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 the set of unprotected locations.

Figure 6 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 technical report includes these omissions. 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.

Checking attacker code. (T-Hole) shows the rule for checking holes in statements. These holes are to be filled by attacker code that can have arbitrary effects on the heap. (T-Hole) states that any property Φ that does not involve the heap is preserved across calls to the attacker code. As such, (T-Hole) is an instance of (T-Frame), which we prove sound under certain syntactic restrictions on the attacker code that fills a hole—roughly, that it be a closed term without any instructions that involve critical types.

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 heaplets 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—this is important since we want our frame rule to say that writes that modify non-Y locations have no effect on the contents of Y-typed objects. If the

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

$$\frac{H \notin \mathit{FV}(\Phi) \quad H \in \Delta}{\Gamma;\Delta \vdash \{\Phi\}\ \bullet_i\ \{\Phi\}}\text{T-Hole} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1, e_2, y\ \text{ok} \quad L = \bigcup_{0 \le i < e_1} \{e_2 + |X|_\Gamma * i\} \quad \mathit{range}_\Gamma\ X = \tau \quad y, X, \mathit{Un}, \tau \in \Delta \\ \sigma_1 = \mathit{copy}_\Gamma\ L\ \mathit{from}\ H\ \mathit{to}\ \bar{X} \quad \Phi, \sigma_2 = \mathit{chkAndRem}_\Gamma\ \tau\ L \quad \sigma_3 = \mathit{updUn}_\Gamma\ L\ \tau\ \bot\end{array}}{\Gamma;\Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := \text{bless}_X[e_1]\ e_2\ \{\Psi\}}\text{T-Bless}$$

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

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

$$\frac{\begin{array}{ccc}\Gamma \vdash e, y\ \text{ok} & y \in \Delta & X \neq \mathit{Un} \\ v_h = \mathit{readFrom}_\Gamma\ H\ (e{:}X) \quad v_x = X_\Gamma(e) & \sigma = [(H_1(e + \mathit{offset}_\Gamma\ X\ p))/y] & \end{array}}{\Gamma;\Delta \vdash \{e \in \mathit{dom}_\Gamma X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ y := X(e).p\ \{\Psi\}}\text{T-Rd} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1, e_2\ \text{ok} \\ H_1 = H[e_1 \leftarrow e_2] \quad \sigma = [H_1/H]\end{array}}{\Gamma; H \vdash \{\sigma(\Psi)\}\ \text{lib}\ e_1 := e_2\ \{\Psi\}}\text{T-LWr}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1, e_2\ \text{ok} \quad X, H \in \Delta \quad X \neq \mathit{Un} \quad v_h = \mathit{readFrom}_\Gamma\ H\ (e_1{:}X) \quad v_x = X_\Gamma(e_1) \\ H_1 = H[(e_1 + \mathit{offset}_\Gamma\ X\ p) \leftarrow e_2] \quad \sigma_1 = \mathit{copy}_\Gamma\ e_1\ \mathit{from}\ H_1\ \mathit{to}\ X \quad \sigma = \sigma_1 \circ [H_1/H]\end{array}}{\Gamma;\Delta \vdash \{e_1 \in \mathit{dom}_\Gamma X \wedge (v_h = v_x \Rightarrow \sigma(\Psi))\}\ X(e_1).p := e_2\ \{\Psi\}}\text{T-Wr} \qquad \frac{\Gamma \vdash e, y\ \text{ok} \quad \sigma = [(H\ e)/y]}{\Gamma; y \vdash \{\sigma(\Psi)\}\ \text{lib}\ y := e\ \{\Psi\}}\text{T-LRd}$$

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

$$\begin{array}{ll}\mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ \mathit{int} & = \cdot \\ \mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ X & = \text{let}\ \hat{v} = \lambda\ell.\mathit{readFrom}_\Gamma\ Y\ (\ell{:}X)\ \text{in} \\ & \quad [(\lambda\ell.\text{if}\ \ell \in L\ \text{then}\ \hat{v}\ \ell\ \text{else}\ X\ \ell)/X] \\ \mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ (\tau_1, \tau_2) & = \text{let}\ L_2 = \{\ell + |\tau_1|_\Gamma \mid \ell \in L\}\ \text{in} \\ & \quad \text{let}\ \sigma_1 = \mathit{copy}_\Gamma\ L\ \mathit{from}\ Y\ \mathit{to}\ \tau_1\ \text{in} \\ & \quad \text{let}\ \sigma_2 = \mathit{copy}_\Gamma\ L_2\ \mathit{from}\ Y\ \mathit{to}\ \tau_2\ \text{in} \\ & \quad \sigma_1 \circ \sigma_2\end{array}$$

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

$$\begin{array}{ll}\mathit{chkAndRem}_\Gamma\ \mathit{int}\ L & = (L \subseteq \mathit{dom}\ \mathit{Un}, \cdot) \\ \mathit{chkAndRem}_\Gamma\ X\ L & = \text{let}\ \Phi = \forall x.x \in L \Rightarrow x \in \mathit{dom}_\Gamma(X)\ \text{in} \\ & \quad (\Phi, [(\lambda\ell.\text{if}\ \ell \in L\ \text{then}\ \bot\ \text{else}\ X\ \ell)/X]) \\ \mathit{chkAndRem}_\Gamma\ (\tau_1, \tau_2)\ L & = \text{let}\ L_2 = \{\ell + |\tau_1|_\Gamma \mid \ell \in L\}\ \text{in} \\ & \quad \text{let}\ \Phi_1, \sigma_1 = \mathit{chkAndRem}_\Gamma\ \tau_1\ L\ \text{in} \\ & \quad \text{let}\ \Phi_2, \sigma_2 = \mathit{chkAndRem}_\Gamma\ \tau_2\ L_2\ \text{in} \\ & \quad (\Phi_1 \wedge \Phi_2, \sigma_1 \circ \sigma_2)\end{array}$$

Membership of types in the modifies set, Δ

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

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

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

Figure 6. $\Gamma;\Delta \vdash \{\Phi\}\ s\ \{\Psi\}$: A Floyd-Hoare logic for YCORE (Selected rules)

contents of an Y object are not first blessed, then a write to a sub-object X can modify the contents of some Y-object, which is inconsistent with the frame rule. To comply with this restriction, the program above first blesses 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 $\mathit{copy}_\Gamma\ L\ \mathit{from}\ H\ \mathit{to}\ X$, we read X-typed tuple values from the heap H at each location in L into the heaplet for 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 $\mathit{Un} : \mathit{int} \to \mathit{int}$ 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, $\mathit{chkAndRem}_\Gamma\ (\mathit{int}, \mathit{int})\ \{\ell\}$ 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 $\mathit{chkAndRem}_\Gamma\ (X, \mathit{int})\ \{\ell\}$ 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 $\mathit{updUn}_\Gamma\ L\ \tau\ \bot$

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 II-B, 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.

Reads and writes. The static semantics of checked reads (T-Rd) and writes (T-Wr) closely mirrors the reduction rules for these constructs in the dynamic semantics. Dynamically, both instructions require the reference being used to be blessed—this manifests itself as a pre-condition in the static semantics that $e \in dom_\Gamma X$. Since the dynamic semantics includes a check to make sure that the value being read or written to is uncorrupted (aborting otherwise), (T-Rd) and (T-Wr) allows us to assume that $v_h = v_x$, *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. The rules (T-LRd) and (T-LWr) provide no special semantics for un-checked reads and writes in the static semantics—libraries are free to read from or write to arbitrary portions of the heap, but leave all critical heaplets unchanged.

D. Soundness and robust safety

The main formal result of this paper is a soundness property for YCORE that is robust even when a program s is composed with attacker programs. We begin by making precise our definition of an attacker that can mount only non-control data attacks.

Definition 1 (Valid attacker program)**.** *A hole-free statement s is a valid attacker program if both of the following conditions are true:*

1) *FV(s) $= \emptyset$, where FV(s) are the free local variables and critical type names in s.*
2) *s does not contain statements of the form (newtype $X = \tau$ in s) or (assert Φ).*

The next lemma establishes that valid attackers are always verifiable in our logic. A corollary of this property is that programs that are verified in our logic remain verifiable even when composed with valid attackers.

Lemma 1 (Valid attackers are trivially verifiable)**.** *For any valid attacker program s, the triple $\Gamma; H \vdash \{True\}\ s\ \{True\}$ is derivable, where $\Gamma = H{:}int \to int, Un{:}int \to int$.*

Proof*: (Sketch) Since s has no free type names and creates no new types, s is free of instructions like $X(e_1).p := e_2$ that involve manipulation of critical data types. So, for any $X \neq H$, X is not in the modifies set. Likewise, s has no free local variables, and hence modifies no local variables. Finally, s is also free of assertions. The remaining statements involve arbitrary reads and write to the heap H, the usual control constructs, and operations on new local variables. Arbitrary combinations of these remaining constructs satisfy the trivial Hoare triple $\{True\}\ s\ \{True\}$.* □

Corollary 2 (Robustness under composition with valid attackers)**.** *For any Γ, Δ, Φ, Ψ, program s with hole $\bullet_i$ and valid attacker program s_i; If $\Gamma; \Delta \vdash \{\Psi\}\ s\ \{\Phi\}$ then $\Gamma; \Delta \vdash \{\Psi\}\ s[s_i]_i\ \{\Phi\}$.*

Note that YCORE provides no first-class control constructs (*e.g.*, computed jumps) thereby preventing attackers from subverting the control flow of the program. Furthermore, although technically feasible in YCORE, we also forbid valid attackers from modifying local variables used by the program since this corresponds conceptually to allowing attackers to modify locations on the stack (which in practice amounts to allowing attackers to modify return addresses stored in stack frames). As such, valid attackers in YCORE are capable of mounting only pure, non-control data attacks.

Finally, we state our soundness result, a theorem that guarantees that verified YCORE programs never get stuck (although they may abort). In the statement below $\vdash \Gamma; \Delta$ ok and $\Gamma \vdash \Psi$ ok are simple well-formedness conditions on the free names of environments and formulas. The relation $\vdash E : \Gamma$ states that the runtime environment E is well-typed according to the bindings in Γ, while the judgment

1. $\Gamma; \mathsf{flag}, H \vdash \{[\mathsf{cgiCmd}, \mathsf{cgiCmd} + |\mathsf{cchar}|_\Gamma * 1024) \in \mathsf{cchar}\}$
   $\mathsf{flag} = \mathsf{CheckRequest}\ (\mathsf{cgiCmd})$
   $\{\mathsf{flag} \neq 0 \Rightarrow \mathtt{validCmd}(\mathsf{cchar}, \mathsf{cgiCmd})\}$
2. $\Gamma; H \vdash \{\mathit{True}\}\ \mathsf{Log}(\texttt{"..."})\ \{\mathit{True}\}$
3. $\Gamma; H \vdash \{\mathtt{validCmd}(\mathsf{cchar}, \mathsf{cgiCmd}) \wedge \mathtt{validDir}(\mathsf{dchar}, \mathsf{cgiDir})\}$
   $\mathsf{ExecuteRequest}(\mathsf{cgiDir}, \mathsf{cgiCmd})$
   $\{\mathit{True}\}$

Figure 7. Three triples to illustrate the power of the frame rule

$E \models \Phi$ is a first-order entailment relation for a formula Φ with free variables bound in E. Clause (1) of part (A) states that the configuration $(E; s)$ is not stuck. Clause (2) states that the new state E' is well-typed in an extension of the environment Γ. Clauses (3) and (4) state that the program s' is verifiable but with the same post-condition, Ψ and a new pre-condition Φ', and with a modifies set that includes at most the variables modifiable by s and possibly any new locals or heaplets allocated in the single step of reduction. Clause (5) ensures that the new pre-conditions Φ' is valid in the new state E'. Finally, part (B) states that when the computation has terminated, the post-condition is valid.

Theorem 3 (Soundness)**.** *For all environments Γ, Δ (such that $\vdash \Gamma; \Delta$ ok); formulas Φ, Ψ (such that $\Gamma \vdash \Psi$ ok); well-formed stores E (such that $\vdash E : \Gamma$) that satisfy the pre-condition ($E \models \Phi$); and hole-free programs s such that $\Gamma; \Delta \vdash \{\Phi\}\ s\ \{\Psi\}$:*

(A) *If $s \neq$ skip, then there exists $E', s', \Gamma', \Phi', \Delta'$ such that all of the following are true:*
   (1) $(E; s) \rightsquigarrow (E'; s')$;
   (2) $\vdash E' : \Gamma, \Gamma'$;
   (3) $\Delta' \subseteq \Delta \cup dom\ \Gamma'$;
   (4) $\Gamma, \Gamma'; \Delta' \vdash \{\Phi'\}\ s'\ \{\Psi\}$; and
   (5) $E' \models \Phi'$,

(B) *If $s =$ skip, then $E \models \Psi$.*

E. The power of the frame rule

This section revisits the `nullhttpd` example of Section II-A and shows how, using our logic, we can reason about the safety of the program. Recall that the example defines two types cchar and dchar, where the static variables cgiCmd and cgiDir hold arrays of these types respectively. The program contains a call to the function ExecuteRequest(cgiDir, cgiCmd), and our goal is to ensure that both arguments to this function are not corrupted, either by buffer overruns within `nullhttpd`, or by the effects of libraries it uses. We can capture this specification by assuming that the three triples in Figure 7 hold for some binary predicates `validCmd` and `validDir`.

These triples are given in a context Γ that includes bindings for the local variable flag and the type names cchar and dchar. The static variables cgiCmd and cgiDir are arbitrary address constants. Additionally, in order to fit in YCORE, we model CheckRequest and ExecuteRequest as inlined sequences of instructions that are free to use arbitrary YCORE instructions. In contrast, Log(`"..."`) represents a sequence of instructions from a library function, whose only effects are via unchecked reads and writes.

The first triple states that the call to CheckRequest modifies the heap and flag, and decides if cgiCmd is a validCmd when it can be shown to be an array of protected cchars. The second triple states that Log can have arbitrary effects on the heap H, since it contains library calls. However, it has no effects on the heaplets corresponding to cchar and dchar. The third triple says that ExecuteRequest demands a pre-condition to ensure that both its arguments are valid.

Our semantics (via Lemma 1) ensures that any sequence s of well-scoped library commands (*e.g.*, the call to Log) satisfies the trivial Hoare triple $\{\mathit{True}\}s\{\mathit{True}\}$ and modifies no type maps X aside from H. In such a case, according to the frame rule, a formula Φ that only references types X and local variables x inaccessible to the library s 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.

To illustrate this power, consider executing our example in a context where $\mathtt{validDir}(\mathsf{dchar}, \mathsf{cgiDir})$ initially holds true. We can guard the call to ExecuteRequest with a test to make sure that flag is non-zero, and verify that the sequence of commands are valid. In particular, using the frame rule, we preserve the predicate $\mathtt{validDir}(\mathsf{dchar}, \mathsf{cgiDir})$ across the first triple, since it only modifies flag and the heap, whereas the free variables of the predicate include only the map for dchar (cgiDir is a constant). Likewise, we preserve both $\mathtt{validDir}(\mathsf{dchar}, \mathsf{cgiDir})$ and the post-condition of the first triple above across the call to Log, without examining the code of Log, even though it has arbitrary effects on the heap.

IV. IMPLEMENTATION

The YARRA compiler is implemented as a plug-in to the CIL compiler infrastructure [22]. 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. These two techniques allow us to run YARRA in two modes. In *whole program protection* mode, we use source protections 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 to protect the backing store, but leaving the library untouched. The remainder of this section describes source and library protections in detail.

A. YARRA *source protections*

YARRA source protections are applied to modules compiled 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 accessed.

The runtime system maintains the type information and implements the checks. The key data structure is a map that associates each memory address with a critical object, if it belongs to one. The runtime system exposes the following functions that manipulate the map.

Bless: **void** bless<ytype t>(**void** *p). The bless function updates the map to reflect that addresses [p, p + **sizeof**(t)) are part of a critical object of type t.

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.

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 addresses [p, p + **sizeof**(t)) in the map of association with t.

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.

B. 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 field to the map described in Section IV-A. Critical writes update this field as well as the value at their target address. The runtime functions are similar to those in Section IV-A, 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.

Program	YARRA Protections	Orig. LOC / Mod. LOC	Bless / Unbless
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 8. YARRA-protected Applications

- ***Bless.*** bless is augmented to copy values of newly blessed addresses to the backing store.

As mentioned earlier, critically-typed writes are also 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. We use hardware page protections to protect the integrity of the backing store. The backing store uses a special critical memory manager (CMM), implemented using the BGET memory manager [32], for memory allocations. The memory pool given to the CMM 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.

V. 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-overhead performance protections.

A. *Hardening server applications with* YARRA

Chen *et. al* identify non-control data attacks on real-world applications, including FTP, SSH, Telnet and HTTP servers. These applications share a common characteristic: they each have a well-defined module that handles a small amount of security-sensitive data i.e., critical data structures. The applications are well-suited to YARRA protections precisely because they share this characteristic.

We show how these applications can be hardened with minimal effort, often with only a new critical type, a few calls to bless and unbless, and minor changes to statements using critical variables. We chose the data-structures to protect in each application based on the attacks described in Chen *et al.*'s paper. Figure 8 shows the server applications we harden, the nature of critical data protected,

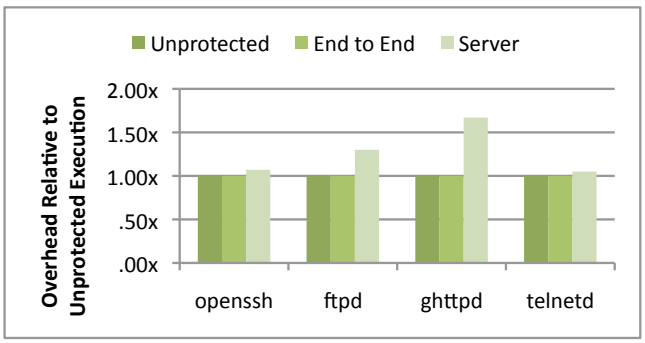

Figure 9. Runtime overhead for hardening data vulnerabilities using YARRA's targeted protection mode, measured from the client ("End to End") and server perspectives. There was no measurable overhead from the client's perspective. A value of *1x* indicates no measurable overhead.

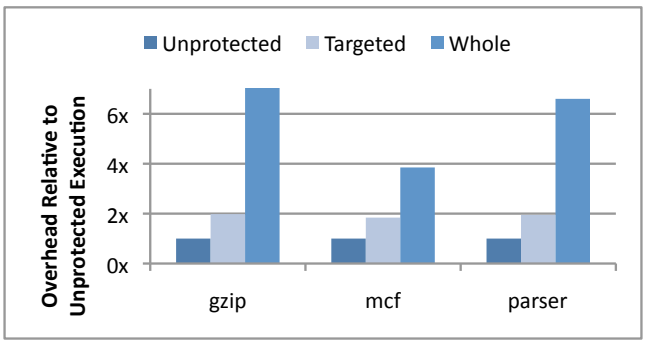

Figure 10. CPU overhead for securing allocator metadata using YARRA's targeted and whole program protection modes. A value of *1x* indicates no measurable overhead.

and the amount of code changed. 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.

We measure end-to-end performance to gauge the impact of applying YARRA protections. For each server, we define a client/server interaction wherein the client connects, performs a small task, and disconnects. By design, each interaction exercises vulnerable code in the server. We compare the run times of a client connecting to vulnerable (unmodified) and hardened servers, normalizing the results against the run time connecting to the vulnerable server. Figure 9 shows our results ("End to End").[2]

We found no measurable overhead between connecting to hardened and vulnerable servers, irrespective of the total lines of code in the program or number of memory accesses throughout the code. YARRA was extremely efficient in protecting the security-critical modules identified by Chen *et al.* as vulnerable to non-control data attacks.

In order to investigate further, we instrumented each server to isolate and collect run-time data from within the protected module, allowing us to measure function slowdown for hardened server functions. Our findings are shown in Figure 9 ("Server"), reflecting a modest performance impact (below 1.6x) on the hardened module.

B. *Stress-testing the performance of* YARRA

We employ a second, atypical use case to evaluate the performance of the YARRA run time under heavy load, wherein we use YARRA to protect module data structures so that clients may not corrupt it. For this study, we experiment with the BGET memory allocator [32], using YARRA to protect BGET's metadata from clients that use the allocator in a way reminiscent of the idealized allocator example presented in Section II-B. The BGET clients we measure are three SPECINT2000 programs also used in the WIT paper [2]. Unlike the server applications of the previous case study, these clients frequently call routines (allocation and deallocation) which contain bless and unbless operations, exercising our implementation vigorously.

Figure 10 illustrates our results,[3] comparing both protection modes discussed in Section IV. We found that targeted protection is much more efficient with these applications, indicating that the cost of boundary crossings from protected code to library code is less than instrumenting every read and write in the application. Even with targeted protection, however, we incur a 2x overhead on the SPEC benchmarks.

There are two bottlenecks in our current implementation, namely read/write instrumentations and boundary crossings. Because our implementation is not as highly optimized as other, similar bounds-checking implementations (e.g. [21], [27]), 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 [4]. Alternatively, the address map structure may be hidden in a separate process, using a technique similar to the one proposed by Berger et al. [3]. These techniques would make boundary crossings take constant time (instead of being linear with the size of the map), albeit at the cost of look-up speed.

Finally, this experiment marks YARRA as a viable modular protection. The changes to BGET were minimal, requiring only 16 calls to bless/unbless and modifying 43 out of 241 lines in total. The SPEC applications did not change at all.

VI. RELATED WORK

Preventing non-control data attacks. Kong et al. [14] propose ensuring data integrity as a special case of taint checking. They separate data and instructions into tainted and taintless, and ensure that each instruction operates on

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

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

the appropriately type of data. They implement their solution with hardware support. Data-flow integrity (DFI) [7] computes data dependencies between instructions using static analysis and ensures that the flow of data at runtime obeys these dependencies. Data Space Randomization (DSR) [5] XORs the contents of memory with a random key, making it difficult for an attacker to correctly subvert the contents. Both DFI and DSR differ from Yarra in that they apply protections to all data (and not just critical data), do not provide language support for partial protection, and do not formalize the semantics of their solutions. SIDAN [10] detects non-control data attacks using techniques from the intrusion detection literature. However, it does not provide any formal guarantees about the protection.

Array bounds checking. Early array bounds-checking techniques (e.g., Jones and Lin [13]) had substantial performance overheads, and more recent work (e.g., [21] 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 explicitly and precisely, and guarantees that all program references, including those performed in external components, do not violate the integrity of such objects.

Dhurjati et al. [11] 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 [31] attempts to isolate the potential negative effects of external components by preventing memory operations and other unwanted interactions, such as system calls, that might be harmful. Castro et al. describe BGI (Byte-Granularity Isolation) [8], 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 [24] also takes the approach of explicitly protecting part of the entire memory state. Like Samurai, YARRA also focuses on protecting critical data from memory corruption 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.

Formal reasoning. The most closely related theories emanate from a line of research started in the 70s with the Euclid programming language [17]. 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 [30] re-examined 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.* [16]. 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.

VII. DISCUSSION

This paper presents YARRA, a lightweight extension to C that allows programmers to protect the integrity of critical data structures in their programs, even in the presence of untrusted third-party libraries. We formalize the key semantic properties of YARRA by developing a sound program logic for it. The logic includes a novel type-based frame rule that gives programmers access to powerful modular reasoning techniques. We show YARRA is effective in practice by protecting important server applications, tens of thousands of lines long, from known vulnerabilities—in each case, we modify at most a few hundred lines of code. Moreover, the end-to-end performance overhead is negligible in the security-centric examples we studied.

We conclude this paper by discussing how YARRA can complement existing protection mechanisms for C programs. One effective protection against control-based attacks is to ensure control-flow integrity (CFI) [1]. Combining CFI with YARRA would give stronger guarantees against both control-based and non-control data attacks than CFI alone. Further, it would require less overhead than combining CFI with complete array bounds checking. While many approaches to array bounds checking have been proposed, none are in widespread use. We believe that this is because of the performance overheads imposed and issues related to whole-program compilation and third-party code — issues where YARRA's alternative design allows it to excel.

We also consider the value of using YARRA in cases where other techniques such as CFI and array bounds

checking are impractical. Specifically, modern systems, such as Microsoft Windows, rely on a collection of techniques to defend against attacks, implemented in the compiler [19], heap [20], and the hardware [18]. While these mechanisms prevent a number of common attack vectors, they do not prevent arbitrary buffer overruns from corrupting either control data (such as vtable pointers) or non-control data (such as passwords). As a result, many publicly available documents demonstrate how to corrupt structures such as the Windows heap metadata to mount a successful attack [25].

In this context, YARRA provides a novel and systematic way to harden applications from attacks. Consider the following scenario: an attacker exploits a buffer overrun in a heap object to overwrite a function pointer in another object or in the heap metadata. Without YARRA, the standard mitigation of this exploit would be to patch the buffer overflow. However, this leaves the program vulnerable to other attacks that overwrite the data, through a different buffer overflow, for example. With YARRA, the mitigation would be to make the function pointer critical, thus protecting the system not just from the one exploit, but from *every* exploit that would attempt to overwrite that function pointer. Note that one does not need to know what vulnerabilities are present or where they are present, in order to deploy YARRA protection.

YARRA would also be effective when used in conjunction with hardware protection such as Data Execution Prevention (DEP), which prevents attackers from injecting code into the heap and jumping to it. Attackers can bypass DEP using return-to-libc attacks and return-oriented programming [6]. However, to do so they still need to overwrite a vulnerable function pointer somewhere in the heap. In such cases, YARRA can be deployed to selectively protect vulnerable function pointers. In future work, we will explore the scalability of employing YARRA in such scenarios.

Acknowledgements: We thank Emery Berger and the anonymous reviewers for useful feedback that helped improve this work. Portions of this material are based upon work supported under NSF grant 1016937 and an NSERC Discovery grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or NSERC.

REFERENCES

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

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

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

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

[5] S. Bhatkar and R. Sekar. Data space randomization. In *DIMVA*, volume 5137, pages 1–22. Springer, 2008.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS 2008*. ACM, 2008.

[7] M. Castro, M. Costa, and T. L. Harris. Securing software by enforcing data-flow integrity. In *OSDI*. USENIX, 2006.

[8] 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.

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

[10] J.-C. Demay, E. Totel, and F. Tronel. SIDAN: A tool dedicated to software instrumentation for detecting attacks on non-control-data. In *CRiSIS*, pages 51–58. IEEE, 2009.

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

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

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

[14] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *ASID*, 2006.

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

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

[17] 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.

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

[19] Microsoft. Gs flag (buffer security check). http://msdn.microsoft.com/en-us/library/8dbf701c%28VS.80%29.aspx, 2005.

[20] Microsoft. Preventing the exploitation of user mode heap corruption vulnerabilities. http://blogs.technet.com/b/srd/archive/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption vulnerabilities.aspx, 2009.

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

[22] 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.

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

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

[25] P. Phantasmagoria. The malloc maleficarum: Glibc malloc exploitation techniques. http://packetstormsecurity.org/files/view/40638/MallocMaleficarum.txt, 2005.

[26] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE, 2002.

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

[28] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Modular protections against non-control data attacks. Technical Report Microsoft Research, TR-2010-158, 2011.

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

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

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

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

APPENDIX

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

Lemma 4 (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 5 (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 6 (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 7 (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 4, 5, and 6. ■

Lemma 8 (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 9 (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 10 (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 9.* ■

Lemma 11 (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 12 (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 13 (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 14 (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 13 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 $\dfrac{x \notin dom\ \Gamma \qquad \Gamma, x; \Delta, x \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma; \Delta \vdash \{\forall x.\Phi\}\ local\ x\ in\ s\ \{\Psi\}}$ *and, from (H5), we have $E \models \forall x.\Phi$.*

By inversion on the reduction relation, we have (H6) $(E; local\ x\ in\ s) \rightsquigarrow (E'; s)$, 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 4 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 $\dfrac{\Gamma \vdash \tau\ ok \qquad X \notin dom\ \Gamma \qquad \hat{\tau} = int \to \tau \qquad \Gamma, X{:}\hat{\tau}; \Delta, X \vdash \{\Phi\}\ s\ \{\Psi\}}{\Gamma; \Delta \vdash \{\forall X{:}\hat{\tau}.X = \lambda\ell.\bot \Rightarrow \Phi\}\ newtype\ X = \tau\ in\ s\ \{\Psi\}}$

By inversion on the reduction relation, we have $(E; newtype\ X = \tau\ in\ s) \rightsquigarrow (E, X \mapsto (\lambda\ell.\bot{:}int \to \tau); s)$
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 $\dfrac{\begin{array}{c}\Gamma \vdash e_1, e_2, y\ ok \qquad L = \bigcup_{0 \le i < e_1} \{e_2 + |X|_\Gamma * i\} \qquad range_\Gamma\ X = \tau \qquad y, X, Un, \tau \in \Delta \\ \sigma_1 = copy_\Gamma\ L\ from\ H\ to\ X \qquad \Phi, \sigma_2 = chkAndRem_\Gamma\ \tau\ L \qquad \sigma_3 = updUn_\Gamma\ L\ \tau\ \bot\end{array}}{\Gamma; \Delta \vdash \{\Phi \wedge (\sigma_1 \circ \sigma_2 \circ \sigma_3 \circ [e_2/y])(\Psi)\}\ y := bless_X[e_1]\ e_2\ \{\Psi\}}$

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)**:*

(H6) is $\dfrac{\begin{array}{c}[\![e_1]\!]_E = n \qquad [\![e_2]\!]_E = \ell \qquad L = \bigcup_{0 \le i < n} \{\ell + |X|_E * i\} \qquad \tau = range_E\ X \\ E_1 = chkAndRem_E\ \tau\ L \qquad E_2 = copy_{E_1}\ L\ from\ H\ to\ X \qquad E' = updUn_{E_2}\ L\ \tau\ \bot\end{array}}{(E; y := bless_X[e_1]\ e_2) \rightsquigarrow (E'[y \mapsto \ell]; skip)}$

For goal (G0), we appeal to Lemmas 10, 11, and 12 to conclude that $E' = E[Un \leftarrow \hat{v}_1][X \leftarrow \hat{v}_2][X_3 \leftarrow \hat{v}_3] \ldots [X_n \leftarrow \hat{v}_n]$, where $X_3, \ldots X_n \in \tau$.
From the premises of (H6), we have $X, Un, y, \tau \in \Delta$, 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 8, 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 10, 11, and 12 we get that $\sigma = [a_1/Un][a_2/X][a_3/X_1] \ldots [a_n/X_n][e_2/y]$ with $\forall i. E \models a_i = \hat{v}_i$
From repeated application of Corollary 7, we get $E' \models \Psi$, as required.

***Case T-UnBless**: Analogous to T-Bless.*

***Case T-Write**:*

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

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

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

For (G0), we use Lemma 10 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 8.
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 $X = Un$ or $E \models X_E(\ell) = \mathit{readFrom}_E\ H\ (e_1{:}X)$.
From the premises of (H4) we have $X \neq Un$.
Thus, we have $E \models \sigma(\Psi)$, and we use Corollary 7 to get $E' \models \Psi$, as required.

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

***Case T-IsX**:*

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

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

***Sub-case E-IsX-Then**:*

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

Goals (G0), (G1) and (G2) are trivial, since the store is unchanged.
For (G3) we show $E \models \Phi_1$, since we have $E \models (X = Un \vee v_h = v_x)) \Rightarrow \Phi_1$ from (H5).
From the premises of (H6) we have $\mathit{inSync}_E\ \ell\ X$, from which we obtain $X = Un$ 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**:*

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

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

***Case T-LibWrite**:*

$$\textit{We have (H4)}\ \frac{\Gamma \vdash e_1, e_2\ ok \quad H_1 = H[e_1 \leftarrow e_2] \quad \sigma = [H_1/H]}{\Gamma; H \vdash \{\sigma(\Psi)\}\ \mathit{lib}\ e_1 := e_2\ \{\Psi\}}$$

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

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

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

For (G3), we show $E' \models \Psi$, *from* $E \vdash \sigma(\Psi)$ *and Corollary 7.*
For (G4), we we use (T-Skip) for $\Gamma'; \Delta \vdash \{\Psi\}\ \mathit{skip}\ \{\Psi\}$.

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 15 (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) $s \neq \mathit{skip}$.

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* $\forall X, l.l \in \mathit{dom}\ X \Rightarrow \mathit{inSync}_E\ X\ l$ *then* $s' \neq \mathit{abort}$.

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	::=	$H \mapsto (\hat{v}{:}\hat{\tau}), \mathit{Un} \mapsto (\hat{v}{:}\hat{\tau}) \mid E, X \mapsto (\hat{v}{:}\hat{\tau}) \mid E, x \mapsto i$	
static env.	Γ	::=	$H{:}\hat{\tau}, \mathit{Un}{:}\hat{\tau} \mid \Gamma, X{:}\hat{\tau} \mid \Gamma, x$	
environment	$\mathcal{E}$	::=	$E \mid \Gamma$	

Figure 11. Syntax

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

$$
\begin{array}{lcll}
X_E(\ell) & = & v & \textbf{when } E(X) = (\hat{v}{:}\tau) \text{ and } \hat{v}\ \ell \rightsquigarrow v \\
X_\Gamma(\ell) & = & X\ \ell & \\
dom_E X & = & \{\ell \mid X_{\mathcal{E}}(\ell) \neq \bot\} & \\
dom_\Gamma X & = & dom\ X & \\
range_E\ X & = & \tau & \textbf{when } E(X) = (\hat{v}{:}int \to \tau) \\
range_\Gamma\ X & = & \tau & \textbf{when } \Gamma(X) = int \to \tau \\
|int|_{\mathcal{E}} & = & 1 & \\
|Y|_{\mathcal{E}} & = & |range_{\mathcal{E}}\ Y|_{\mathcal{E}} & \\
|(\tau_1, \tau_2)|_{\mathcal{E}} & = & |\tau_1|_{\mathcal{E}} + |\tau_2|_{\mathcal{E}} & \\
\mathit{offset}_{\mathcal{E}}\ \tau\ \cdot & = & 0 & \\
\mathit{offset}_{\mathcal{E}}\ (\tau_1, \tau_2)\ 0p & = & \mathit{offset}_{\mathcal{E}}\ \tau_1\ p & \\
\mathit{offset}_{\mathcal{E}}\ (\tau_1, \tau_2)\ 1p & = & |\tau_1|_{\mathcal{E}} + \mathit{offset}_{\mathcal{E}}\ \tau_2\ p & \\
\mathit{offset}_{\mathcal{E}}\ Y\ p & = & \mathit{offset}_{\mathcal{E}}\ (range_{\mathcal{E}}\ Y)\ p & \\
\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}int) & = & Y_{\mathcal{E}}(\ell) & \\
\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}Z) & = & \mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}(range_{\mathcal{E}}\ Z)) & \\
\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}(\tau_1, \tau_2)) & = & (\mathit{readFrom}_{\mathcal{E}}\ Y\ (\ell{:}\tau_1), \mathit{readFrom}_{\mathcal{E}}\ Y\ ((\ell + |\tau_1|_E){:}\tau_2)) & \\
\mathit{notBlessed}_{\mathcal{E}}\ \ell & = & \ell \in dom_{\mathcal{E}} \mathit{Un} &
\end{array}
$$

Figure 12. 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 $: (Env * Locs * Map * Type) \to Subst$

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

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

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

Membership of types in Δ ***and of map variables in types***

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

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

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

Figure 13. Auxiliary functions used in static semantics only (Reproduced from Figure 6)

$E[X \leftarrow \hat{v}] = E_1, X \mapsto (\hat{v}{:}\hat{\tau}), E_2$ **when** $E = E_1, X \mapsto (\hat{v}{:}\hat{\tau}), E_2$

$\mathit{blessed}_E\ L\ X = \forall \ell \in L.\ell \in \mathit{dom}_E X$

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

$\mathit{inSync}_E\ \ell\ \mathit{Un} = \mathit{True}$

$\mathit{inSync}_E\ \ell\ X = X_E(\ell) = \mathit{readFrom}_E\ H\ (\ell{:}X)$ **when** $X \neq \mathit{Un}$

$\mathit{inSync}_E\ L\ X = \forall \ell \in L.X_E(\ell) = \mathit{readFrom}_E\ H\ (\ell{:}X)$ **when** $X \neq \mathit{Un}$

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

$\mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ \mathit{int} = E$

$\mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ X = E[X \leftarrow (\lambda\ell.\text{if } \ell \in L \text{ then } \mathit{readFrom}_E\ Y\ (\ell{:}X) \text{ else } X(\ell))]$

$\mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ (\tau_1, \tau_2) =$ let $E_1 = \mathit{copy}_E\ L\ \mathit{from}\ Y\ \mathit{to}\ \tau_1$ in
let $L_1 = \{\ell + |\tau_1|_{E_1} \mid \ell \in L\}$ in
$\mathit{copy}_{E_1}\ L_1\ \mathit{from}\ Y\ \mathit{to}\ \tau_2$

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

$\mathit{chkAndRem}_E\ X\ L = notSync$ **when** $\mathit{blessed}_{\mathcal{E}}\ L\ X \wedge \neg \mathit{inSync}_E\ L\ X$

$\mathit{chkAndRem}_E\ X\ L = E[X \leftarrow \lambda\ell.\text{if } \ell \in L \text{ then } \bot \text{ else } (X\ell)]$ **when** $\mathit{blessed}_{\mathcal{E}}\ L\ X \wedge \mathit{inSync}_E\ L\ X$

$\mathit{chkAndRem}_E\ \mathit{int}\ L = E$ **when** $L \subseteq \mathit{dom}_E \mathit{Un}$

$\mathit{chkAndRem}_E\ (\tau_1, \tau_2)\ L =$ let $E_1 = \mathit{chkAndRem}_E\ \tau_1\ L$ in
let $L_1 = \{\ell + |\tau_1|_{E_1} \mid \ell \in L\}$ in
$\mathit{chkAndRem}_{E_1}\ \tau_2\ L_1$

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

$\mathit{updUn}_E\ L\ \mathit{int}\ \hat{e} = E[\mathit{Un} \leftarrow \lambda\ell.\text{if } \ell \in L \text{ then } \hat{e} \text{ else } \mathit{Un}\ \ell]$

$\mathit{updUn}_E\ L\ X\ \hat{e} = E$

$\mathit{updUn}_E\ L\ (\tau_1, \tau_2)\ \hat{e} =$ let $E_1 = \mathit{updUn}_E\ L\ \tau_1\ \hat{e}$ in
let $L_1 = \{\ell + |\tau_1|_{E_1} \mid \ell \in L\}$ in
$\mathit{updUn}_{E_1}\ L_1\ \tau_2\ \hat{e}$

Figure 14. Auxiliary functions used in dynamic semantics only

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 15. Dynamic semantics of YCORE

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma;\Delta \vdash \{\Phi_1\}\ s_2\ \{\Psi\} \quad \Gamma;\Delta \vdash \{\Phi\}\ s_1\ \{\Phi_1\}}{\Gamma;\Delta \vdash \{\Phi\}\ s_1; s_2\ \{\Psi\}}\ \text{T-Seq}$$

$$\frac{}{\Gamma;\Delta \vdash \{\Psi\}\ \text{skip}\ \{\Psi\}}\ \text{T-Skip} \qquad \frac{}{\Gamma;\Delta \vdash \{\mathit{True}\}\ \text{abort}\ \{\Psi\}}\ \text{T-Abort}$$

Figure 16. A Floyd-Hoare logic for YCORE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 17. 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 18. Interpretation of terms and formulas