CHECKEDCBOX: Formalizing RLBox in Checked C for Incremental Spatial Memory Safety (Extended Version)

This is an extended version of a paper that appears at the 2022 Computer Security Foundations Symposium.

1. Introduction

Vulnerabilities due to memory corruption, especially spatial memory corruptions, are still a major issue for C programs [2, 41] despite a large body of work that tries to prevent them [38]. Vulnerabilities due to memory corruption, especially spatial memory corruptions, are still a major issue for C programs [2, 41] despite a large body of work that tries to prevent them [38]. One way to handle this is to convert these programs into safe languages such as Java. However, it requires considerable development effort. Furthermore, the performance overhead prohibits them from being used in performance-critical settings such as Operating Systems.

With the increased support for high-performant safe system languages such as Rust and Go, we can have tools that convert C to a memory-safe language, e.g., Rust, a promising memory-safe systems language [26]. Unfortunately, Rust is too different from C to constitute a practical target [43], at least for now: the (best of breed) `c2rust` tool [3, 17] transliterates C to *unsafe, non-idiomatic* Rust, and recent work [9] is able to take only small steps to close the safety gap and still requires considerable work from developers.

There are other retrofitting techniques such as CCured [31], Softbound [27], Low Fat pointers [6], and Address Sanitizer (ASAN) [36] aim to enforce spatial safety automatically *without any developer effort*, by analyzing a C program and compiling it to include run-time safety checks. Unfortunately, the resulting run-time overhead is too high for deployment (between 60%-200%), especially in low-powered IoT devices where the usage of C-based system software (e.g., FreeRTOS) is still rampant. Figure **??** shows the summary of these existing approaches.

Checked C [40] as a practical target to which to convert legacy/active C code. Checked C extends C with *checked pointer types* which are restricted by the compiler to spatially safe uses (temporal safety is underway [46]). Such pointers have one of three possible types, `ptr`<T>, `array_ptr`<T>, or `nt_array_ptr`<T> (ptr, arr, and $ntarr$ for short), representing a pointer to a single element, array of elements, or null-terminated array of elements of type T, respectively. The latter two have an associated *bounds annotation*; e.g., a declaration `array_ptr`<int> `p` : `count(n)` says that `p` is a pointer to an `int` array whose size is `n`. Checked C's Clang/LLVM-based compiler represents checked pointers as system-level memory words, i.e., without "fattening" metadata, ensuring backward compatibility. The compiler uses these bounds annotations to add dynamic checks prior to checked pointer accesses, to prevent spatial safety violations. These run-time checks can often be proved redundant and removed by LLVM, yielding good performance. Tarditi et al. [40] reported average run-time overheads of 8.6% on a small benchmark suite, and [5] found essentially no overhead when running Checked C-converted portions of the FreeBSD kernel.

However, converting existing C programs to Checked C requires considerable effort [5]. Recently, Machiry et al. developed 3C [21] that tries to automatically convert a program to Checked C by adding appropriate pointer annotations. However, as described in 3C, complete automated conversion is infeasible and consequently requires the developer to convert some code regions manually. Thanks to the backward compatibility of Checked C, the partially annotated program still enjoys the benefits of spatial memory safety on those regions using only Checked pointers (i.e., checked or safe regions). But, the unconverted code regions (or unsafe regions) can affect pointers in safe regions and violate certain assumptions leading to vulnerabilities, as demonstrated by cross-language attacks [24]. Although the blameless proof exists [35], it does not state that spatial safety violations cannot happen in Checked regions but rather states that Checked regions *cannot be blamed for any spatial safety violations*. For instance, a checked function expects a pointer to a buffer of five elements, but an unchecked region can call this function with a pointer to a buffer of three elements. This results in a spatial safety violation in the Checked region, but of course, the blame or the root cause is in the unchecked region. Furthermore, since checked and unchecked regions execute in the same address space, spatial memory corruptions in unchecked regions can take down the complete program despite having checked regions. We need an isolation mechanism to ensure that code executed as part of unchecked regions does not violate the safety guarantees in checked regions. Such an isolation mechanism should have low/no overhead and be backward compatible (i.e., pointers should have no additional metadata), preserving the advantages of Checked C.

In this work, we propose CHECKEDCBOX, an isolation mechanism that enables unchecked regions or, in general, any arbitrary set of functions to be executed in an isolated address space ($uc - region$). So that checked regions can

still enjoy true spatial memory safety even in the presence of unchecked or unsafe regions. CHECKEDCBOX extends Checked c using **tainted** (`t_*`) types, representing the code and data that belong to the $uc-region$.

Automated Partitioning. A pointer (checked or unchecked) can be marked as tainted; similarly, a function can be marked as tainted. Along with enforcing spatial safety constraints of checked tainted pointers, the CHECKEDCBOX compiler will bundle all the tainted types (functions and pointers) so that they remain in an isolated sandbox ($uc-region$), and non-tainted types will remain in the process address space ($c-region$).

Memory Access Restrictions. The functions in the $uc-region$ can only access tainted pointers, i.e., memory within the corresponding sandbox region. However, the functions in $c-region$ have no-such restrictions and can access both c- and uc-region's memory. Similarly, the functions in $c-region$ can invoke any function in $uc-region$ but not the other way around, except for call-back functions, which we will discuss later.

Marshaling free communication. As functions in the $uc-region$ can only access tainted types, $c-region$ functions should use tainted types to pass pointer arguments to $uc-region$ functions. We avoid marshaling as all tainted buffers (i.e., regions pointed by tainted pointers) are allocated in $uc-region$ and thus can be accessed in both c- and uc-region.

Additional checks. Although memory isolation prevents direct violations, $uc-region$ code can still affect $c-region$ through tainted pointers by confused deputy attacks. However, our compiler avoids these attacks by ensuring, either statically or through dynamic checks, that tainted pointers in $c-region$ can only point to $uc-region$ address space, and our type system enforces that tainted pointers cannot be assigned to non-tainted pointers.

2. Overview and Transcendence

This section discusses the key merge features between Checked-C and RLBox and the problems we are trying to solve.

Maintaining Non-crashing. Previously, the main guarantee of Checked C [20] was the blame theorem, i.e., if there is a crash, the source is from some unchecked regions. For example, at Figure 3 line 31, we call a unchecked function `f` with a checked null-terminated array (NT-array) pointer argument. At line 8, depending on the NT-array size, `free(s[10])` might crash. Even if it does not crash, line 38 is doomed because of the `free` call.

The philosophy is that unchecked regions are written in C and unstable, and programmers will eventually remove them by converting them to Checked C code. The reality is that users might not want to convert everything. For some insignificant or handy history C code, such as library functions, they might want to keep it as long as no crashing.

The sources of crashing in Checked C are (1) unchecked regions crash themselves; and (2) the misuse of checked pointers in unchecked regions. Enlightened by RLBox [29], we sandbox the unchecked regions and then we utilize the Checked C type system to disallow checked pointers to be used in an unchecked region. To achieve the communication between checked and unchecked regions, we create tainted pointers that can be shared by different regions, whose data are stored in the sandboxed heap. Users are required to copy data to tainted pointers that are shared in unchecked regions. For example, we copy the checked pointer data to the tainted pointer `tp` at Figure 3 line 26, and input the tainted pointer to the unchecked function at line 34. At line 8, even if statement might crash, since tainted pointers are stored in the sandboxed heap, it can be recovered. At line 37, the use of a tainted pointer in a checked region requires a verification on it. This is handled by inserting additional checks and creating exception handling before the use by the CHECKEDCBOX compiler. Thus, the checked pointer `p` is safely used at line 38.

```
1  //in checked region
2
3  int compare_1(nt_array_ptr<char> x: count (0),
4    nt_array_ptr<char> y : count (0)) {
5    int len_x = strlen(x);
6    int len_y = strlen(y);
7    return sum(x,len_x) < sum(y,len_y);
8    }
9  ...
10
11 int stringsort(
12  nt_array_ptr<nt_array_ptr<char>> s : count (n),
13  ptr<(int)(nt_array_ptr<char>,
14    nt_array_ptr<char>)> cmp, int n) {
15   int i, j, gap;
16   int didswap;
17
18   for(gap = n / 2; gap > 0; gap /= 2) {
19     {
20     do  {
21         didswap = 0;
22         for(i = 0; i < n - gap; i++)
23            {
24              j = i + gap;
25              if((*cmp)(s[i], s[j]) > 0)
26              {
27                 int len = strlen(s[i]);
28                 nt_array_ptr<char>
29                   tmp : count (len) = s[i];
30                 s[i] = s[j];
31                 s[j] = tmp;
32                 didswap = 1;
33              }
34            }
35         } while(didswap);
36     }
37   }
38
39   return 0;
40 }
```

Figure 1: Checked stringsort Code

```
//in checked region, tainted version
int tainted_compare_1(
  nt_array_tptr<char> x : count (0),
  nt_array_tptr<char> y : count (0)) {
 checked (x,y) {
  int len_x = strlen(x);
  int len_y = strlen(y);
  nt_array_ptr<char> tx : count (len_x)
   = malloc(nt_array<char>, len_x);
  nt_array_ptr<char> ty : count (len_y)
   = malloc(nt_array<char>, len_y);
  safe_memcpy(tx,x,len_x);
  safe_memcpy(ty,y,len_y);
  return compare_1(tx,ty);
 }
}
...

//calling the function turns
//an unchecked region to a checked region.
int tainted_stringsort(nt_array_tptr
     <nt_array_tptr<char>> s : count (n),
 tptr<(int)(nt_array_tptr<char>,
  nt_array_tptr<char>)> cmp, int n) {
 checked (s,cmp,n) {
  int i;
  nt_array_ptr<nt_array_ptr<char>> p : count (n)
    = malloc(nt_array<nt_array_ptr<char>>, n);
  for(i = 0; i < n; i++) {
    int len = strlen s[i];
    nt_array_ptr<char> tmp : count (len)
      = new malloc(nt_array<char>, len);
    safe_memcpy(tmp,s[i],len);
    p[i] = tmp;
  }
  ptr<(int)(nt_array_ptr<char> : count (0),
  nt_array_ptr<char> : count (0))>
    cfun = find_check(cmp);

  return stringsort(p,cfun);
 }
}
```

Figure 2: Tainted stringsort Code

In short, banning checked pointer uses in unchecked regions and providing tainted shared pointers are the solution for crashing. In CHECKEDCBOX, we proved the non-crashing theorem based on the CHECKEDCBOX type system, .i.e, any program execution is stopped in a predictable manner where either the execution terminates or is stopped at a place where an exception handler exists. In fact, our work is *the first formalism of RLBox*, and formally shows the security guarantee that RLBox provides is actually the non-crashing theorem.

Formalism Function Pointers. In C, manipulating function pointers is a major way of implementing high order functions. Previously, we assumed all function calls are through calling by name to a global map. In CHECKEDCBOX, we formalize function pointers and maintain the CHECKEDCBOX type soundness. To the best of our knowledge, this is the first work of formalizing C function pointers with security guarantee.

```
//in unchecked region
int f(char ** s, int (*cmp)(char *,char *),
  int (*sort)(char **, int (*)(char *,char *),
              int), int n) {
...

  int i = sort(s,cmp,n);
  free(s[10]);
...
}

int g(int (*cmp)(char *,char *)) {
...
  int real_addr = derandomize(cmp);
...
}

int main(int n) {
  nt_array_ptr<nt_array_ptr<char>> p : count(n)
    = malloc(nt_array<nt_array_ptr<char>>, n);

  nt_array_tptr<nt_array_tptr<char>>
     tp : count(n) =
      tmalloc(nt_array<nt_array_tptr<char>>, n);
...
  safe_memcpy(tp, p, n);

  unchecked {
    if (BAD) // a flag to call different funs.
      //input checked pointers
      f(p, compare_1, stringsort);
    else
      //input tainted pointers
      f(tp, tainted_compare_1,
          tainted_stringsort);
  }

  if (!BAD) safe_memcpy(p, tp, n);
  p[10] = "crash?";

  unchecked {
    if (BAD) g(compare_1)
      else g(tainted_compare_1);
  }

  return 0;
}
```

Figure 3: Tainted Pointer Usage in Calling Unchecked Fun

Figure 1 defines a string sorting algorithm depending on the input function pointer `cmp` defining the generic order for strings, and `compare_1` is an example `cmp` function that adds the ASCII numbers of characters in the two strings and compare the results. In addition, function pointers enable the callback mechanism, i.e., a server sends a function pointer to a client in an unchecked region, and allows the client to access some server resources by calling back the pointer. This is a common usage between a web-browser and

untrusted third party libraries. The function call in Figure 3 line 34 is one such usage.

We also utilize CHECKEDCBOX subtyping relation to permit function pointer static auto-casting. Function pointer type information might contain array pointer bound information, for which it is inconvenient to coincide the defined types for a function implementation and the function pointer type. For example, the `cmp` argument in `stringsort` (Figure 1) has type `ptr<(int)(nt_array_ptr<char>, nt_array_ptr<char>)>`, meaning that the function takes two NT-array pointers with arbitrary size and outputs an integer. The function `compare_1`'s pointer has type `ptr<(int)(nt_array_ptr<char> : count (0), nt_array_ptr<char> : count (0))>` To use `compare_1` in `stringsort`, the type is auto-cast to the `cmp`'s type. In general, if function pointer x has type $*(tl \to t)$, and y has $*(tl' \to t')$, in order to use x as y, tl' should be a subtype of tl and t subtypes to t'.

Not Exposing Checked Pointer Addresses. The design for the non-crashing property bans the checked pointer uses in unchecked regions. Thus, there is no reason to permit checked pointer variable assignments in unchecked regions; especially, this might expose a checked pointer address to untrusted parties. For example, the call to function g at Figure 3 line 41 lives in an unchecked region, and g might use some mechanism, such as derandomizing ASLR [37], to achieve the checked pointer address. Thus, it enables a third party to access any checked heap and function data by simple pointer arithmetic.

To prevent the checked pointer address leak, we prevent any unchecked regions acknowledge checked pointer variables. In addition, to facilitate checked function callbacks, the CHECKEDCBOX compiler compiles every checked function with an additional tainted shell function. Users are required to serve unchecked regions with the tainted shell pointer instead of the original checked function pointer. For example, `tainted_compare_1` and `tainted_stringsort` in Figure 2 are the tainted shells of the checked functions `compare_1` and `stringsort`. In the tainted shells, the arguments are tainted versions of the corresponding arguments in the checked functions. Inside the shell body, create checked pointer copies of the tainted arguments, and call the checked functions. In addition, once a checked function returns, if the output is a checked pointer, we also its the data to a new tainted pointer and exit the shell. Figure 3 line 42 is an example of serving the function call living in an unchecked region with a tainted shell pointer argument `tainted_compare_1`. Even if g derndomizes its address (line 14), the shell address is in the sandbox and has no harm, and calling the shell never exposes any checked pointer information outside of the shell.

Conceptually, the shell is run in a checked region. One can imagine that a tainted shell is a safe closure that contains a checked block. Once the closure is called, the system is turned to a checked region, so that the checked function call at Figure 2 line 14 is safe, even if it is called by g in Figure 3 , because it lives in the checked region. In the CHECKEDCBOX formalism, we formalize a *checked* block

Variables: x Integers: $n ::= \mathbb{Z}$

Context Mode:	m	::=	$\mathtt{c} \mid \mathtt{u}$
Pointer Mode:	ξ	::=	$m \mid \mathtt{t}$
Bound:	b	::=	$n \mid x + n$
	β	::=	(b, b)
Word Type:	τ	::=	$\mathtt{int} \mid \mathtt{ptr}^{\xi}\ \omega$
Type Flag:	κ	::=	$nt \mid \cdot$
Type:	ω	::=	$\tau \mid [\beta\ \tau]_{\kappa} \mid \forall\ \overline{x}.\ \overline{\tau} \to \tau$
Expression:	e	::=	$n : \tau \mid x \mid e + e \mid (\tau)e \mid \langle\tau\rangle e$
		|	$\mathtt{strlen}(x) \mid *e \mid *e = e$
		|	$\mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{if}\ (e)\ e\ \mathtt{else}\ e$
		|	$\mathtt{malloc}(\xi, \omega) \mid e(\overline{e})$
		|	$\mathtt{unchecked}(\overline{x})\{e\} \mid \mathtt{checked}(\overline{x})\{e\}$

Figure 4: CORECHKCBOX Syntax

$$m \vdash \mathtt{int} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \mathtt{ptr}^{\xi}\ [\beta\ \tau]_{\kappa}}$$

$$\frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \mathtt{ptr}^{\xi}\ \tau} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m \qquad FV(\overline{\tau}) \cup FV(\tau) \subseteq \overline{x}}{m \vdash \mathtt{ptr}^{\xi}\ (\forall\ \overline{x}.\ \overline{\tau} \to \tau)}$$

$$\mathtt{t} \wedge \mathtt{c} = \mathtt{u} \qquad \xi \wedge \mathtt{u} = \mathtt{u} \qquad \mathtt{c} \wedge m = m \qquad m_1 \wedge m_2 = m_2 \wedge m_1$$
$$\xi \leq \xi \qquad \mathtt{t} \leq \xi$$

Figure 5: Well-formedness for Types

on top of the existing unchecked regions, and the transition of a tainted shell call creates a checked block containing the shell body. We also make sure that no arguments in these tainted shell contains any checked pointers, as well as no output is of a checked type.

3. Formalization

This section describes the formal model of CHECKEDCBOX, called CORECHKCBOX, making precise its syntax, semantics, and type system. It also develops CORECHKCBOX's meta-theory, including type soundness, the non-exposure, and the non-crashing theorem.

3.1. Syntax

The syntax of CORECHKCBOX is given by the expression-based language presented in Fig. 4.

There are two notions of type in CORECHKCBOX. Types τ classify word-sized values including integers and pointers, while types ω classify multi-word values such as arrays, null-terminated arrays, functions, and single-word-size values. Pointer types ($\mathtt{ptr}^{m}\ \omega$) include a mode annotation (m) which is either checked (`c`) or unchecked (`u`) and a type (ω) denoting valid values that can be pointed to. Array

types include both the type of elements (τ) and a bound (β) comprised of an upper and lower bound on the size of the array ((b_l, b_h)). Bounds b are limited to integer literals n and expressions $x + n$. Whether an array pointer is null terminated or not is determined by annotation κ, which is nt for null-terminated arrays, and $\cdot$ otherwise (we elide the $\cdot$ when writing the type). CHECKEDCBOX function types ($\forall\ \overline{x}.\ \overline{\tau} \to \tau$) reflect its dependent function declrations, where $\overline{x}$ represents a list of `int`type variables in a dependent function header, which bind all bounds appearing in $\overline{\tau}$ and τ. We have a well-formed requirement for a function type, such that all variables in $\overline{\tau}$ and τ are bounded by $\overline{x}$. Here is the corresponding Checked C syntax for these types:

$$\texttt{array\_tptr<}\tau\texttt{>} : \texttt{count}(n) \quad \Leftrightarrow \quad \texttt{ptr}^{\texttt{t}}\ [(0,n)\ \tau]$$
$$\texttt{nt\_array\_ptr<}\tau\texttt{>} : \texttt{count}(n) \quad \Leftrightarrow \quad \texttt{ptr}^{\texttt{c}}\ [(0,n)\ \tau]_{nt}$$
$$\texttt{tptr<(int)(nt\_array\_tptr<}\tau\texttt{>} : \texttt{count}(n),\ \texttt{nt\_array\_tptr<}\tau\texttt{>)>: count}(n)\texttt{)>}$$
$$\Leftrightarrow\ \texttt{ptr}^{\texttt{t}}\ (\forall\ n.\ \texttt{ptr}^{\texttt{t}}\ [(0,n)\ \tau]_{nt} \times \texttt{ptr}^{\texttt{t}}\ [(0,n)\ \tau]_{nt} \to \texttt{int})$$

As a convention we write $\texttt{ptr}^{\texttt{c}}\ [b\ \tau]$ to mean $\texttt{ptr}^{\texttt{c}}\ [(0,b)\ \tau]$, so the above examples could be rewritten $\texttt{ptr}^{\texttt{c}}\ [n\ \tau]$ and $\texttt{ptr}^{\texttt{c}}\ [n\ \tau]_{nt}$, respectively.

CORECHKCBOX expressions include literals ($n : \tau$), variables (x), addition (e_1 + e_2), static casts ($(\tau)e$), dynamic casts ($\langle\tau\rangle e$) [1], the `strlen` operation ($\texttt{strlen}(x)$), pointer dereference and assignment ($*e$) and ($*e_1 = e_2$), resp.), let binding ($\texttt{let}\ x = e_1\ \texttt{in}\ e_2$), conditionals ($\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2$), memory allocation ($\texttt{malloc}(\xi, \omega)$), function calls ($e(\overline{e})$), unchecked blocks ($\texttt{unchecked}(\overline{x})\{e\}$), and checked blocks ($\texttt{checked}(\overline{x})\{e\}$).

Integer literals n are annotated with a type τ which can be either `int`, or $\texttt{ptr}^m\ \omega$ in the case n is being used as a heap address (this is useful for the semantics); $0 : \texttt{ptr}^m\ \omega$ (for any m and ω) represents the `null` pointer, as usual. The `strlen` expression operates on variables x rather than arbitrary expressions to simplify managing bounds information in the type system; the more general case can be encoded with a `let`. We use a less verbose syntax for dynamic bounds casts; e.g., the following

`dyn_bounds_cast<array_ptr<`τ`>>(`e`, count(`n`))`

becomes $\langle\texttt{ptr}^{\texttt{c}}\ [n\ \tau]\rangle e$.

Compared to the former Checked C model [20], there are four differences. First, the CHECKEDCBOX type annotations have well-formed restrictions in Figure 5, for maintaining non-exposure. Mainly, in a nested pointer $\texttt{ptr}^{\xi}\ (...\texttt{ptr}^{\xi'}\ \tau...)$, if the outside mode ξ is `t` or `u`, the inside mode ξ' cannot be `c`. It is worth noting that pointer modes are a three point partial order ($\leq$), where `t` is the infimum, and $\xi \wedge m$ is a special meet operation that projects pointer modes onto context modes, such that a pointer mode `t` is projected as `u`. Second, $\texttt{malloc}(\xi, \omega)$ includes a mode flag ξ for allocating different pointers in different heap mode regions. For simplicity, we disallow ω to be a function type ($\forall\ \overline{x}.\ \overline{\tau} \to \tau$), Third, the first expression e in a function call ($e(\overline{e})$) represents a function pointer. Fourth, we update $\texttt{unchecked}(\overline{x})\{e\}$ to specify $\overline{x}$ restricting all free variables

1. assumed at compile-time and verified at run-time, see Appendix A

$$\begin{array}{lcl}
\mu & ::= & n:\tau \\
e & ::= & \ldots \mid \texttt{ret}(x, \mu, e) \\
r & ::= & e \mid \texttt{null} \mid \texttt{bounds} \\
E & ::= & \square \mid E + e \mid n:\tau + E \mid (\tau)E \mid \langle\tau\rangle E \mid *E \mid *E = e \\
 & & \mid *n:\tau = E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ (E)\ e\ \texttt{else}\ e \\
 & & \mid E(\overline{e}) \mid n:\tau(\overline{E}) \mid \texttt{unchecked}(\overline{x})\{E\} \mid \texttt{checked}(\overline{x})\{E\}
\end{array}$$

$$\frac{m = \mathit{mode}(E) \qquad e = E[e'] \qquad (\varphi, \mathcal{H}, e') \longrightarrow (\varphi', \mathcal{H}', e'')}{(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', E[e''])}$$

$$\frac{m = \mathit{mode}(E) \qquad e = E[\texttt{if}\ (*x)\ e_1\ \texttt{else}\ e_2] \qquad (\varphi, \mathcal{H}, \texttt{if}\ (*x)\ e_1\ \texttt{else}\ e_2) \longrightarrow (\varphi', \mathcal{H}', e')}{(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', E[e'])} \quad [\mathit{prefer}]$$

$$\frac{\texttt{u} = \mathit{mode}(E) \qquad e = E[e'] \qquad \tau = \mathit{type}(e')}{(\varphi, \mathcal{H}, e) \longrightarrow_{\texttt{u}} (\varphi, \mathcal{H}, E[0:\tau])}$$

$$\begin{array}{l}
\mathit{mode}(E) = \mathit{mode}'(E, \texttt{c}) \\
\mathit{mode}'(\square, m) = m \\
\mathit{mode}'(\texttt{unchecked}(\overline{x})\{E\}, m) = \mathit{mode}'(E, \texttt{u}) \\
\mathit{mode}'(\texttt{checked}(\overline{x})\{E\}, m) = \mathit{mode}'(E, \texttt{c}) \\
\mathit{mode}'(\alpha(E), m) = \mathit{mode}'(E, m) \quad [\mathit{owise}]
\end{array}$$

Figure 6: CORECHKCBOX Semantics: Evaluation

appearing in e. Additionally, we create `checked` blocks indicating the context mode change from `u` to `c`. Both the free variables in the `unchecked` and `checked` blocks and the return types cannot be `c` mode for maintaining non-exposure, i.e., no checked pointer can be communicated between `unchecked` and `checked` blocks.

CORECHKCBOX aims to be simple enough to work with, but powerful enough to encode realistic Checked C idioms. For example, mutable local variables can be encoded as immutable locals that point to the heap; the use of `&` can be simulated with `malloc`; and loops can be encoded as recursive function calls. `structs` are not in Fig. 4 for space reasons, but they are actually in our model, and developed in Appendix F. C-style `unions` have no safe typing in Checked C, so we omit them. Additional syntactic descrition is listed in the Checked C formalism paper [20].

3.2. Semantics

The operational semantics for CORECHKCBOX is defined as a small-step transition relation with the judgment $(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', r)$. Here, φ is a *stack* mapping from variables to values $n : \tau$ and $\mathcal{H}$ is a *heap* that is partitioned into two parts (`c` and `u` regions), each of which maps addresses (integer literals) to values $n:\tau$. If a pointer has mode `c`, it lives in the `c` region; otherwise, it lives in the `u` region. We wrote $\mathcal{H}(m, n)$ to retrieve the n-location heap value in the m region, and $\mathcal{H}(m)[n \mapsto \mu]$ to update location n with the value μ. It is worth noting that CHECKEDCBOX is not a fat-pointer system; thus, in every heap update, the value type annotation remains the same through program

$$m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{int} \qquad m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{ptr}^{\texttt{u}}\ \omega$$

$$m;\Theta;\mathcal{H};\sigma \vdash 0 : \texttt{ptr}^{\xi}\ \omega \qquad \frac{(n{:}\texttt{ptr}^{\xi}\ \omega) \in \sigma}{m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{ptr}^{\xi}\ \omega}$$

$$\frac{\texttt{ptr}^{\xi'}\ \omega' \sqsubseteq_{\Theta} \texttt{ptr}^{\xi}\ \omega \qquad m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{ptr}^{\xi'}\ \omega'}{m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{ptr}^{\xi}\ \omega}$$

$$\frac{\xi \leq m \qquad \Xi(m,n) = \tau\ (\overline{x'}{:}\overline{\tau})\ (\xi, e) \qquad \overline{x} = \{x | (x : \texttt{int}) \in (\overline{x'} : \overline{\tau})\}}{m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{ptr}^{\xi}\ (\forall\ \overline{x}.\ \overline{\tau} \to \tau)}$$

$$\frac{\neg \mathit{fun\_t}(\omega) \qquad \xi \leq m \qquad \forall i \in [0, \mathit{size}(\omega))\ .\ m;\Theta;\mathcal{H};(\sigma \cup \{(n : \texttt{ptr}^{\xi}\ \omega))\} \vdash \mathcal{H}(m, n+i)}{m;\Theta;\mathcal{H};\sigma \vdash n : \texttt{ptr}^{\xi}\ \omega}$$

$$\mathit{fun\_t}(\forall\ \overline{x}.\ \overline{\tau} \to \tau) = \texttt{true} \qquad \mathit{fun\_t}(\omega) = \texttt{false}\ \ [\mathit{owise}]$$

Figure 7: Verification/Type Rules for Constants

executions. Additionally, for both stack and heap, we ensure $FV(\tau) = \emptyset$ for all the value type annotations τ.

While heap bindings can change, stack bindings are immutable—once variable x is bound to $n : \tau$ in φ, that binding will not be updated; we can model mutable stack variables as pointers into the mutable heap. As mentioned, value $0 : \tau$ represents a `null` pointer when τ is a pointer type; correspondingly, $\mathcal{H}(0)$ should always be undefined. The relation steps to a *result* r, which is either an expression or a `null` or `bounds` failure, representing a null-pointer dereference or out-of-bounds access, respectively. Such failures are a *good* outcome; stuck states (non-value expressions that cannot transition to a result r) characterize undefined behavior. The context mode m indicates whether the stepped redex within e was in a checked (`c`) or unchecked (`u`) region.

The rules for the main operational semantics judgment—*evaluation*—are given at the bottom of Fig. 6. The first rule takes an expression e, decomposes it into an *evaluation context* E and a sub-expression e' (such that replacing the hole $\square$ in E with e' would yield e), and then evaluates e' according to the *computation* relation $(\varphi, \mathcal{H}, e') \longrightarrow (\varphi, \mathcal{H}, e'')$, whose rules are given in Fig. 8, discussed shortly. The second rule handles conditionals `if` $(*x)$ e_2 `else` e_3 in redex position specially, delegating directly to the S-IfNTT computation rule, which supports bounds widening; we discuss this rule shortly. When the second and first rules could both apply, we always prefer the second. The third rule describes the exception handling for possible crashing behaviors in unchecked region. A program in `u` mode can non-deterministically crash and the CheckedCBox sandbox mechanism recorvers the program to a safe point $(0{:}\tau)$ and continues with the existing program state.

The *mode* function determines the context mode when evaluating e' based on the context E. For any program execution, the top context mode always starts with `c` ($\mathit{mode}(E) = \mathit{mode}'(E, \texttt{c})$). The result context mode is depeneded on where the $\square$ locates. If it occurs within $(\texttt{unchecked}(\overline{x})\{E'\})$ inside E without a surrounding `checked` block, then the mode is `u`; if the exact opposite happens (within $(\texttt{checked}(\overline{x})\{E'\})$ without surrounding `unchecked` blocks), the mode is `c`; the other cases ($\mathit{mode}'(\alpha(E), m) = \mathit{mode}'(E, m)$) are recursively defined by the above base cases (α represents any syntactic categories in Figure 6 other than $\square$, `unchecked`, and `checked`). Evaluation contexts E define a standard left-to-right evaluation order. (We explain the $\texttt{ret}(x, \mu, e)$ syntax shortly.)

Fig. 8 shows selected rules for the computation relation; we explain them with the help of the example in Figure 1 and Figure 2.

Checked and Tainted Pointer Operations. The rules for pointer related operations—S-DefC, S-DefT, S-AssignArrC, S-AssignArrT, S-DefNull, S-IfNTTC and S-Cast. The first five are deference and assignment operations—illustrate how the semantics checks bounds. Rule S-DefNull transitions attempted null-pointer dereferences to `null`, whereas S-DefC dereferences a `c`-mode non-null (single) pointer. When `null` is returned by the computation relation, the evaluation relation halts the entire evaluation with `null` (using a rule not shown in Fig. 6); it does likewise when `bounds` is returned (see Appendix C).

S-AssignArrC assigns to an array as long as 0 (the point of dereference) is within the bounds designated by the pointer's annotation and strictly less than the upper bound. For the assignment rule, arrays are treated uniformly whether they are null-terminated or not (κ can be $\cdot$ or nt)—the semantics does not search past the current position for a null terminator. The program can widen the bounds as needed, if they currently precede the null terminator: S-DefNTArray, which dereferences an NT array pointer, allows an upper bound of 0, since the program may read, but not write, the null terminator. A separate rule (not shown) handles normal arrays.

Casts. Static casts of a literal $n : \tau'$ to a type τ are handled by S-Cast. In a type-correct program, such casts are confirmed safe by the type system. To evaluate a cast, the rule updates the type annotation on n. Before doing so, it must "evaluate" any variables that occur in τ according to their bindings in φ. For example, if τ was $\texttt{ptr}^{\texttt{c}}\ [(0, x+3)\ \texttt{int}]$, then $\varphi(\tau)$ would produce $\texttt{ptr}^{\texttt{c}}\ [(0, 5)\ \texttt{int}]$ if $\varphi(x) = 2$.

Dynamic casts are accounted for by S-DynCast and S-DynCastBound. In a type-correct program, such casts are assumed correct by the type system, and later confirmed by the semantics. As such, a dynamic cast will cause a `bounds` failure if the cast-to type is incompatible with the type of the target pointer, as per the $n'_l > n_l \vee n_h > n'_h$ condition in S-DynCastBound. An example use of dynamic casts is given on line 7 in Fig. 9. The values of `x` and `n` might not be known statically, so the type system cannot confirm that `x` $\leq$ `n`; the dynamic cast assumes this inequality holds, but then checks it at run-time.

Binding and Function Calls. The semantics handles variable scopes using the special `ret` form. S-Let eval-

S-DEFC
$$\frac{\mathcal{H}(\mathtt{c}, n) = n_a : \tau_a}{(\varphi, \mathcal{H}, *n : \mathtt{ptr}^{\mathtt{c}}\ \tau) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}$$

S-ASSIGNARRC
$$\frac{\mathcal{H}(\mathtt{c}, n) = n_a : \tau_a \qquad 0 \in [n_l, n_h)}{(\varphi, \mathcal{H}, *n : \mathtt{ptr}^{\mathtt{c}}\ [(n_l, n_h)\ \tau]_\kappa = n_1 : \tau_1) \longrightarrow (\varphi, (\mathcal{H}, \mathtt{c})[n \mapsto n_1 : \tau_a], n_1 : \tau)}$$

S-DEFT
$$\frac{\mathcal{H}(\mathtt{u}, n) = n_a : \tau_a \qquad \mathtt{u}; \emptyset; \mathcal{H}; \emptyset \vdash n_a : \tau}{(\varphi, \mathcal{H}, *n : \mathtt{ptr}^{\mathtt{t}}\ \tau) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}$$

S-ASSIGNARRT
$$\frac{\mathcal{H}(\mathtt{u}, n) = n_a : \tau_a \qquad 0 \in [n_l, n_h) \qquad \mathtt{u}; \emptyset; \mathcal{H}; \emptyset \vdash n_1 : \tau}{(\varphi, \mathcal{H}, *n : \mathtt{ptr}^{\mathtt{t}}\ [(n_l, n_h)\ \tau]_\kappa = n_1 : \tau_1) \longrightarrow (\varphi, (\mathcal{H}, \mathtt{u})[n \mapsto n_1 : \tau_a], n_1 : \tau)}$$

S-DEFNULL
$$(\varphi, \mathcal{H}, *0 : \mathtt{ptr}^{\mathtt{c}}\ \omega) \longrightarrow (\varphi, \mathcal{H}, \mathtt{null})$$

S-IFNTTC
$$\frac{\varphi(x) = n : \mathtt{ptr}^{\mathtt{c}}\ [(n_l, 0)\ \tau]_{nt} \qquad \mathcal{H}(\mathtt{c}, n) \neq 0}{(\varphi, \mathcal{H}, \mathtt{if}\ (*x)\ e_1\ \mathtt{else}\ e_2) \longrightarrow (\varphi[x \mapsto n : \mathtt{ptr}^{\mathtt{c}}\ [(n_l, 1)\ \tau]_{nt}], \mathcal{H}, e_1)}$$

S-CAST
$$(\varphi, \mathcal{H}, (\tau) n : \tau') \longrightarrow (\varphi, \mathcal{H}, n : \varphi(\tau))$$

S-LET
$$(\varphi, \mathcal{H}, \mathtt{let}\ x = n : \tau\ \mathtt{in}\ e) \longrightarrow (\varphi[x \mapsto n : \tau], \mathcal{H}, \mathtt{ret}(x, \varphi(x), e))$$

S-RETCON
$$\frac{\varphi(x) = \mu \qquad (\varphi[x \mapsto \mu'], \mathcal{H}, e) \longrightarrow (\varphi', \mathcal{H}', e')}{(\varphi, \mathcal{H}, \mathtt{ret}(x, \mu', e)) \longrightarrow (\varphi'[x \mapsto \mu], \mathcal{H}', \mathtt{ret}(x, \varphi'(x), e'))}$$

S-RETEND
$$(\varphi, \mathcal{H}, \mathtt{ret}(x, \mu, n : \tau)) \longrightarrow (\varphi, \mathcal{H}, n : \tau)$$

S-UNCHECKED
$$(\varphi, \mathcal{H}, \mathtt{unchecked}(\overline{x})\{n : \tau\}) \longrightarrow (\varphi, \mathcal{H}, n : \tau)$$

S-FUNC
$$\frac{\Xi(\mathtt{c}, n) = \tau\ (\overline{x} : \overline{\tau})\ (\mathtt{c}, e)}{(\varphi, \mathcal{H}, n : (\mathtt{ptr}^{\mathtt{c}}\ \tau)(\overline{n_a} : \overline{\tau_a})) \longrightarrow (\varphi, \mathcal{H}, \mathtt{let}\ \overline{x} = \overline{n} : (\overline{\tau}[\overline{n}/\overline{x}])\ \mathtt{in}\ (\tau[\overline{n}/\overline{x}])e)}$$

S-CHECKED
$$(\varphi, \mathcal{H}, \mathtt{checked}(\overline{x})\{n : \tau\}) \longrightarrow (\varphi, \mathcal{H}, n : \tau)$$

S-FUNT
$$\frac{\Xi(\mathtt{u}, n) = \tau\ (\overline{x} : \overline{\tau})\ (\mathtt{t}, e) \qquad \mathtt{u}; \emptyset; \mathcal{H}; \emptyset \vdash n : \mathtt{ptr}^{\mathtt{t}}\ \tau}{(\varphi, \mathcal{H}, n : (\mathtt{ptr}^{\mathtt{t}}\ \tau)(\overline{n_a} : \overline{\tau_a})) \longrightarrow (\varphi, \mathcal{H}, \mathtt{let}\ \overline{x} = \overline{n} : (\overline{\tau}[\overline{n}/\overline{x}])\ \mathtt{in}\ (\tau[\overline{n}/\overline{x}])e)}$$

Figure 8: CORECHKCBOX Semantics: Computation (Selected Rules)

```
1  nt_array_ptr<char> safe_strcat
2     (nt_array_ptr<char> dst : count(n),
3      nt_array_ptr<char> src : count(0), int n) {
4    int x = strlen(dst);
5    int y = strlen(src);
6    nt_array_ptr<char> c : count(n) =
7      dyn_bounds_cast
8            <nt_array_ptr<char>>(dst,count(n));
9      // sets c == dst with bound n (not x)
10   if (x+y < n) {
11     for (int i = 0; i < y; ++i)
12       *(c+x+i) = *(src+i);
13     *(c+x+y) = '\0';
14     return dst;
15   }
16   return null;
17 }
```

Figure 9: Implementation of safe `strcat`

uates to a configuration whose stack is φ extended with a binding for x, and whose expression is $\mathtt{ret}(x, \varphi(x), e))$ which remembers x was previously bound to $\varphi(x)$; if it had no previous binding, $\varphi(x) = \bot$. Evaluation proceeds on e until it becomes a literal $n : \tau$, in which case S-RET restores the saved binding (or $\bot$) in the new stack, and evaluates to $n : \tau$.

Function calls are handled by S-FUN. Recall that array bounds in types may refer to in-scope variables; e.g., parameter `dst`'s bound `count(n)` refers to parameter `n` on lines 2-3 in Fig. 9. A call to function f causes f's definition to be retrieved from Ξ, which maps function names to forms $\tau\ (\overline{x} : \overline{\tau})\ e$, where τ is the return type, $(\overline{x} : \overline{\tau})$ is the parameter list of variables and their types, and e is the function body. The call is expanded into a `let` which binds parameter variables $\overline{x}$ to the actual arguments $\overline{n}$, but annotated with the parameter types $\overline{\tau}$ (this will be safe for type-correct programs). The function body e is wrapped in a static cast $(\tau[\overline{n}/\overline{x}])$ which is the function's return type but with any parameter variables $\overline{x}$ appearing in that type substituted with the call's actual arguments $\overline{n}$. To see why this is needed, suppose that `safe_strcat` in Fig. 9 is defined to return a `nt_array_ptr<int>:count(n)` typed term, and assume that we perform a `safe_strcat` function call as `x=safe_strcat(a,b,10)`. After the evaluation of `safe_strcat`, the function returns a value with type `nt_array_ptr<int>:count(10)` because we substitute bound variable `n` in the defined return type with `10` from the function call's argument list. Note that the S-FUN rule replaces the annotations $\overline{\tau_a}$ with $\overline{\tau}$ (after instantiation) from the function's signature. Using $\overline{\tau_a}$ when executing the body of the function has no impact on the soundness of CORECHKCBOX, but will violate Theorem 4, which we introduce in Sec. 4.

Bounds Widening. Bounds widening occurs when branching on a dereference of a NT array pointer, or when performing `strlen`. The latter is most useful when assigned to a local variable so that subsequent code can use the result,

e.g., e in `let` x `=` `strlen`(y) `in` e. Lines 4 and 5 in Fig. 9 are examples. The widened upper bound precipitated by `strlen`(y) is extended beyond the lifetime of x, as long as y is live. For example, `x`'s scope in line 4 at runtime is the whole function body in `safe_strcat` because the lifetime of the pointer `dst` is in the function body. This is different from the Checked C specification, which only allows bound widening to happen within the scope of `x`, and restoring old bound values once `x` dies. We allow widening to persist outside the scope at run-time as long as we are within the stack frame, and we show this does not necessarily require the use of fat pointers in Sec. 4.

Rule S-StrWiden implements `strlen` widening. The predicate $\forall i.n \leq i < n + n_a \Rightarrow (\exists n_i\ t_i.\mathcal{H}(n+i) = n_i : \tau_i \wedge n_i \neq 0))$ aims to find a position $n+n_a$ in the NT array that stores a null character, where no character as indexes between n and $n+n_a$ contains one. (This rule handles the case when $n_a > n_h$, the $n_a \leq n_h$ case is handled by a normal `strlen` rule; see Appendix C.)

Rule S-IfNTT performs bounds widening on x when the dereference $*x$ is not at the null terminator, but the pointer's upper bound is 0 (i.e., it's at the end of its known range). x's upper bound is incremented to 1, and this count persists as long as x is live. For example, `s`'s increment (lines 5–8) is live until the return of the function in Fig. **??**; thus, line 11 is valid because `s`'s upper bound is properly extended.

3.3. Typing

We now turn to the CoreChkCBox type system. The typing judgment has the form $\Gamma; \Theta \vdash_m e : \tau$, which states that in a type environment Γ (mapping variables to their types) and a predicate environment Θ (mapping integer-typed variables to Boolean predicates), expression e will have type τ if evaluated in mode m. Key rules for this judgment are given in Fig. 10. In the rules, $m \leq m'$ uses the two-point lattice with `u` $<$ `c`. All remaining rules are given in Appendix B and E.

Pointer Access. Rules T-DefArr and T-AssignArr type-check array dereference and assignment operations resp., returning the type of pointed-to objects; rules for pointers to single objects are similar. The condition $m \leq m'$ ensures that unchecked pointers can only be dereferenced in unchecked blocks; the type rule for `unchecked`$(e)\{\}$ sets $m =$ `u` when checking e. The rules do not attempt to reason whether the access is in bounds; this check is deferred to the semantics.

Casting and Subtyping. Rule T-Cast rule forbids casting to checked pointers when in checked regions (when $m =$ `c`), but τ is unrestricted when $m =$ `u`. The T-CastCheckedPtr rule permits casting from an expression of type τ' to a checked pointer when $\tau' \sqsubseteq \texttt{ptr}^{\texttt{c}}\ \tau$. This subtyping relation $\sqsubseteq$ is given in Fig. 11; the many rules ensure the relation is transitive. Most of the rules handle casting between array pointer types. The second rule $0 \leq b_l \wedge b_h \leq 1 \Rightarrow \texttt{ptr}^m\ \tau \sqsubseteq \texttt{ptr}^m\ [(b_l, b_h)\ \tau]$ permits treating a singleton pointer as an array pointer with $b_h \leq 1$ and $0 \leq b_l$.

Since bounds expressions may contain variables, determining assumptions like $b_l \leq b_l'$ requires reasoning about those variables' possible values. The type system uses Θ to make such reasoning more precise.[2] Θ is a map from variables x to predicates P, which have the form $P ::= \top \mid \texttt{ge\_0}$. If Θ maps x to $\top$, that means that the variable can possibly be any value; `ge_0` means that $x \geq 0$. We will see how Θ gets populated and give a detailed example of subtyping below.[3]

Rule T-DynCast typechecks dynamic casting operations, which apply to array pointer types only. The cast is accepted by the type system, as its legality will be checked by the semantics.

Bounds Widening. The bounds of NT array pointers may be widened at conditionals, and calls to `strlen`. Rule T-If handles normal branching operations; rule T-IfNT is specialized to the case of branching on $*x$ when x is a NT array pointer whose upper bound is 0. In this case, true-branch e_1 is checked with x's type updated so that its upper bound is incremented by 1; the else-branch e_2 is type-checked under the existing assumptions. For both rules, the resulting type is the join of the types of the two branches (according to subtyping). This is important for the situation when x itself is part of the result, since x will have different types in the two branches.

Rule T-Str handles the case for when `strlen`(y) does not appear in a let binding. Rule T-LetStr handles the case when it does, and performs bounds widening. The result of the call is stored in variable x, and the type of y is updated in Γ when checking the let-body e to indicate that x is y's upper bound. Notice that the lower bound b_l is unaffected by the call to `strlen`(y); this is sound because we know that `strlen` will always return a result n such that $n \geq b_h$, the current view of x's upper bound. The type rule tracks `strlen`'s widened bounds within the scope of x, while the bound-widening effect in the semantics applies to the lifetime of y. Our type preservation theorem in Sec. 3.4 shows that our type system is a sound model of the CoreChkCBox semantics, and we discuss how we guarantee that the behavior of our compiler formalization and the semantics matches in Sec. 4.

This rule also extends Θ when checking e, adding a predicate indicating that $x \geq 0$. To see how this information is used, consider this example. The `return` on line 14 of Fig. 9 has an implicit static cast from the returned expression to the declared function type (see rule T-Fun, described below). In type checking the `strlen` on line 4, we insert a predicate in Θ showing `x` ≥ 0. The static cast on line 14 is valid according to the last line in Fig. 11:

2. Technically, the subtyping relation $\sqsubseteq$ and the bounds ordering relation $\leq$ are parameterized by Θ; this fact is implicit to avoid clutter.

3. As it turns out, the subtyping relation is also parameterized by φ, which is needed when type checking intermediate results to prove type preservation; source programs would always have $\varphi = \emptyset$. Details are in Appendix D.

$$\text{T-DefArr}\quad \frac{m \le m' \qquad \Gamma;\Theta \vdash_m e : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa}{\Gamma;\Theta \vdash_m *e : \tau}$$

$$\text{T-AssignArr}\quad \frac{\Gamma;\Theta \vdash_m e_1 : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad \Gamma;\Theta \vdash_m e_2 : \tau' \qquad \tau' \sqsubseteq \tau \qquad m \le m'}{\Gamma;\Theta \vdash_m *e_1 = e_2 : \tau}$$

$$\text{T-Unchecked}\quad \frac{\Gamma;\Theta \vdash_u e : \tau}{\Gamma;\Theta \vdash_m \texttt{unchecked}(e)\{:\}\tau}$$

$$\text{T-Cast}\quad \frac{m = \texttt{c} \Rightarrow \tau \neq \texttt{ptr}^{\texttt{c}}\ \tau'' \text{ for any } \tau'' \qquad \Gamma;\Theta \vdash_m e : \tau'}{\Gamma;\Theta \vdash_m (\tau)e : \tau}$$

$$\text{T-CastCheckedPtr}\quad \frac{\Gamma;\Theta \vdash_m e : \tau' \qquad \tau' \sqsubseteq \texttt{ptr}^{\texttt{c}}\ \tau}{\Gamma;\Theta \vdash_m (\texttt{ptr}^{\texttt{c}}\ \tau)e : \texttt{ptr}^{\texttt{c}}\ \tau}$$

$$\text{T-DynCast}\quad \frac{\Gamma;\Theta \vdash_m e : \texttt{ptr}^{m}\ [\beta'\ \tau]_\kappa}{\Gamma;\Theta \vdash_m \langle \texttt{ptr}^{m}\ [\beta\ \tau]_\kappa \rangle e : \texttt{ptr}^{m}\ [\beta\ \tau]_\kappa}$$

$$\text{T-If}\quad \frac{\Gamma;\Theta \vdash_m e : \tau \qquad \Gamma;\Theta \vdash_m e_1 : \tau_1 \qquad \Gamma;\Theta \vdash_m e_2 : \tau_2}{\Gamma;\Theta \vdash_m \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 : \tau_1 \sqcup \tau_2}$$

$$\text{T-IfNT}\quad \frac{\Gamma;\Theta \vdash_m x : \texttt{ptr}^{c}\ [(b_l, 0)\ \tau]_{nt} \qquad \Gamma[x \mapsto \texttt{ptr}^{c}\ [(b_l, 1)\ \tau]_{nt}];\Theta \vdash_m e_1 : \tau_1 \qquad \Gamma;\Theta \vdash_m e_2 : \tau_2}{\Gamma;\Theta \vdash_m \texttt{if}\ (*x)\ e_1\ \texttt{else}\ e_2 : \tau_1 \sqcup \tau_2}$$

$$\text{T-Str}\quad \frac{\Gamma;\Theta \vdash_m e : \texttt{ptr}^{m}\ [\beta\ \tau_a]_{nt}}{\Gamma;\Theta \vdash_m \texttt{strlen}(e) : \texttt{int}}$$

$$\text{T-LetStr}\quad \frac{\Gamma(y) = \texttt{ptr}^{\texttt{c}}\ [(b_l, b_h)\ \tau_a]_{nt} \qquad \Gamma[x \mapsto \texttt{int}, y \mapsto [\texttt{ptr}^{\texttt{c}}\ [(b_l, x)\ \tau_a]_{nt}];\Theta[x \mapsto \texttt{ge\_0}] \vdash_m e : \tau}{\Gamma;\Theta \vdash_m \texttt{let}\ x = \texttt{strlen}(y)\ \texttt{in}\ e : \tau[b_h/x]}$$

$$\text{T-Let}\quad \frac{x \in FV(\tau') \Rightarrow e_1 \in \text{Bound} \qquad \Gamma;\Theta \vdash_m e_1 : \tau \qquad \Gamma[x \mapsto \tau];\Theta \vdash_m e_2 : \tau'}{\Gamma;\Theta \vdash_m \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau'[e_1/x]}$$

$$\text{T-Fun}\quad \frac{\Xi(f) = \tau\ (\overline{x} : \overline{\tau})\ e \qquad \Gamma;\Theta \vdash_m \overline{e} : \overline{\tau'} \qquad \overline{\tau'} \sqsubseteq \overline{\tau}[\overline{e}/\overline{x}]}{\Gamma;\Theta \vdash_m f(\overline{e}) : \tau[\overline{e}/\overline{x}]}$$

$$\text{T-Ret}\quad \frac{\Gamma(x) \neq \bot \qquad \Gamma;\Theta \vdash_m e : \tau}{\Gamma;\Theta \vdash_m \texttt{ret}(x, \mu, e) : \tau}$$

Figure 10: Selected type rules

$$\begin{aligned}
\tau &\sqsubseteq \tau \\
0 \le b_l \land b_h \le 1 \Rightarrow \texttt{ptr}^{m}\ \tau &\sqsubseteq \texttt{ptr}^{m}\ [(b_l, b_h)\ \tau] \\
b_l \le 0 \land 1 \le b_h \Rightarrow \texttt{ptr}^{m}\ [(b_l, b_h)\ \tau] &\sqsubseteq \texttt{ptr}^{m}\ \tau \\
b_l \le 0 \land 1 \le b_h \Rightarrow \texttt{ptr}^{m}\ [(b_l, b_h)\ \tau]_{nt} &\sqsubseteq \texttt{ptr}^{m}\ \tau \\
b_l \le b_l' \land b_h' \le b_h \Rightarrow \texttt{ptr}^{m}\ [(b_l, b_h)\ \tau]_{nt} &\sqsubseteq \texttt{ptr}^{m}\ [(b_l', b_h')\ \tau] \\
b_l \le b_l' \land b_h' \le b_h \Rightarrow \texttt{ptr}^{m}\ [(b_l, b_h)\ \tau]_{\kappa} &\sqsubseteq \texttt{ptr}^{m}\ [(b_l', b_h')\ \tau]_{\kappa}
\end{aligned}$$

Figure 11: Subtyping

$$\texttt{ptr}^{c}\ [(0, x)\ \tau]_\kappa \sqsubseteq \texttt{ptr}^{c}\ [(0, 0)\ \tau]_\kappa$$

because $0 \le 0$ and $0 \le x$, where the latter holds since Θ proves $x \ge 0$. Without Θ, we would need a dynamic cast.

In our formal presentation, Θ is quite simple and is just meant to illustrate how static information can be used to avoid dynamic checks; it is easy to imagine richer environments of facts that can be leveraged by, say, an SMT solver as part of the subtyping check [33, 42].

Dependent Functions and Let Bindings. Rule T-Fun is the standard dependent function call rule. It looks up the definition of the function in the function environment Ξ, type-checks the actual arguments $\overline{e}$ which have types $\overline{\tau'}$, and then confirms that each of these types is a subtype of the declared type of f's corresponding parameter. Because functions have dependent types, we substitute each parameter e_i for its corresponding parameter x_i in both the parameter types and the return type. Consider the `safe_strcat` function in Fig. 9; its parameter type for `dst` depends on `n`. The T-Fun rule will substitute `n` with the argument at a call-site.

Rule T-Let types a `let` expression, which also admits type dependency. In particular, the result of evaluating a `let` may have a type that refers to one of its bound variables (e.g., if the result is a checked pointer with a variable-defined bound); if so, we must substitute away this variable once it goes out of scope. Note that we restrict the expression e_1 to syntactically match the structure of a Bounds expression b (see Fig. 4).

Rule T-Ret types a `ret` expression, which does not appear in source programs but is introduced by the semantics when evaluating a let binding (rule S-Let in Fig. 8); this rule is needed for the preservation proof. After the evaluation of a let binding a variable x concludes, we need to restore any prior binding of x, which is either $\bot$ (meaning that there is no x originally) or some value $n{:}\tau$.

3.4. Type Soundness and Blame

In this subsection, we focus on our main meta-theoretic results about CoreChkCBox: type soundness (progress and preservation) and blame. These proofs have been carried out in our Coq model, found at https://github.com/plum-umd/checkedc.

The type soundness theorems rely on several notions of *well-formedness*:

Definition 1 (Type Environment Well-formedness). A type environment Γ is well-formed iff every variable

mentioned as type bounds in Γ are bounded by `nat` typed variables in Γ.

Definition 2 (Heap Well-formedness). A heap $\mathcal{H}$ is well-formed iff (i) $\mathcal{H}(0)$ is undefined, and (ii) for all $n{:}\tau$ in the range of $\mathcal{H}$, type τ contains no free variables.

Definition 3 (Stack Well-formedness). A stack snapshot φ is well-formed iff for all $n{:}\tau$ in the range of φ, type τ contains no free variables.

We also need to introduce a notion of *consistency*, relating heap environments before and after a reduction step, and type environments, predicate sets, and stack snapshots together.

Definition 4 (Stack Consistency). A type environment Γ, variable predicate set Θ, and stack snapshot φ are consistent—written $\Gamma;\Theta \vdash \varphi$—iff for every variable x, $\Theta(x)$ is defined implies $\Gamma(x) = \tau$ for some τ and $\varphi(x) = n{:}\tau'$ for some n, τ' where $\tau' \sqsubseteq \tau$.

Definition 5 (Stack-Heap Consistency). A stack snapshot φ is consistent with heap $\mathcal{H}$—written $\mathcal{H} \vdash \varphi$—iff for every variable x, $\varphi(x) = n{:}\tau$ implies $\mathcal{H};\emptyset \vdash n : \tau$.

Definition 6 (Heap-Heap Consistency). A heap $\mathcal{H}'$ is consistent with $\mathcal{H}$—written $\mathcal{H} \triangleright \mathcal{H}'$—iff for every constant n, $\mathcal{H};\emptyset \vdash n : \tau$ implies $\mathcal{H}';\emptyset \vdash n : \tau$.

Moreover, as a program evaluates, its expression may contain literals $n{:}\tau$ where τ is a pointer type, i.e., n is an index in $\mathcal{H}$ (perhaps because n was chosen by `malloc`). The normal type-checking judgment for e is implicitly parameterized by $\mathcal{H}$, and the rules for type-checking literals confirm that pointed-to heap cells are compatible with (subtypes of) the pointer's type annotation; in turn this check may precipitate checking the type consistency of the heap itself. We follow the same approach as Ruef et al. [34], and show the rules in Fig. 7; the judgment $\mathcal{H};\sigma \vdash n : \tau$ is used to confirm literal well-typing, where σ is a set of pointer literals already checked in $\mathcal{H}$ (to allow pointer cycles). See Appendix B for further discussion.

Progress now states that terms that don't reduce are either values or their mode is unchecked:

Theorem 1 (Progress).
For any Checked C program e, heap $\mathcal{H}$, stack φ, type environment Γ, and variable predicate set Θ that are all are well-formed, consistent ($\Gamma;\Theta \vdash \varphi$ and $\mathcal{H} \vdash \varphi$) and well typed ($\Gamma;\Theta \vdash_{\mathtt{c}} e : \tau$ for some τ), one of the following holds:

- e is a value ($n{:}\tau$).
- there exists φ' $\mathcal{H}'$ r, such that $(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', r)$.
- $m = \mathtt{u}$, or there exists E and e', such that $e = E[e']$ and $\mathit{mode}(E) = \mathtt{u}$.

Proof: By induction on the typing derivation.

Preservation states that a reduction step preserves both the type and consistency of the program being reduced.

Theorem 2 (Preservation). For any Checked C program e, heap $\mathcal{H}$, stack φ, type environment Γ, and variable predicate set Θ that are all are well-formed, consistent ($\Gamma;\Theta \vdash \varphi$ and $\mathcal{H} \vdash \varphi$) and well typed ($\Gamma;\Theta \vdash_{\mathtt{c}} e : \tau$ for some τ), if there exists φ', $\mathcal{H}'$ and e', such that $(\varphi, \mathcal{H}, e) \longrightarrow_{\mathtt{c}} (\varphi', \mathcal{H}', e')$, then $\mathcal{H}'$ is consistent with $\mathcal{H}$ ($\mathcal{H} \triangleright \mathcal{H}'$) and there exists Γ', Θ' and τ' that are well formed, consistent ($\Gamma';\Theta' \vdash \varphi'$ and $\mathcal{H}' \vdash \varphi'$) and well typed ($\Gamma';\Theta' \vdash_{\mathtt{c}} e : \tau'$), where $\tau' \sqsubseteq \tau$.

Proof: By induction on the typing derivation.

Using these two theorems we can prove our main result, *blame*, which states that if a well-typed program is *stuck*—expression e is a non-value that cannot take a step[4]—the cause must be the (past or imminent) execution of code in an unchecked region.

Theorem 3 (The Blame Theorem). For any Checked C program e, heap $\mathcal{H}$, stack φ, type environment Γ, and variable predicate set Θ that are well-formed and consistent ($\Gamma;\Theta \vdash \varphi$ and $\mathcal{H} \vdash \varphi$), if e is well-typed ($\varphi;\Theta \vdash_{\mathtt{c}} e : \tau$ for some τ) and there exists φ_i, $\mathcal{H}_i$, e_i, and m_i for $i \in [1,k]$, such that $(\varphi, \mathcal{H}, e) \longrightarrow_{m_1} (\varphi_1, \mathcal{H}_1, e_1) \longrightarrow_{m_2} \ldots \longrightarrow_{m_k} (\varphi_k, \mathcal{H}_k, r)$ and r is *stuck*, then there exists $j \in [1,k]$, such that $m_j = \mathtt{u}$, or there exists E and e', such that $r = E[e']$ and $\mathit{mode}(E) = \mathtt{u}$.

Proof: By induction on the number of steps of the Checked C evaluation ($\longrightarrow_m^*$), using progress and preservation to maintain the invariance of the assumptions.

Compared to Ruef et al. [34], proofs for CoreChkCBox were made challenging by the addition of dependently typed functions and dynamic arrays, and the need to handle bounds widening for NT array pointers. These features required changes in the runtime semantics (adding a stack, and dynamically changing bounds) and in compile-time knowledge of them (to soundly typing widened bounds).

4. Compilation

The semantics of CoreChkCBox uses annotations on pointer literals in order to keep track of array bounds information, which is used in premises of rules like S-DefArray and S-AssignArr to prevent spatial safety violations. However, in the real implementation of Checked C, which extends Clang/LLVM, these annotations are not present—pointers are represented as a single machine word with no extra metadata, and bounds checks are not handled by the machine, but inserted by the compiler.

This section shows how CoreChkCBox annotations can be safely erased: using static information a compiler can insert code to manage and check bounds metadata, with no loss of expressiveness. We present a compilation algorithm that converts from CoreChkCBox to CoreC, an untyped language without metadata annotations. The syntax and semantics CoreC closely mirrors that of CoreChkCBox; it differs only in that literals lack type annotations and its

4. Note that `bounds` and `null` are *not* stuck expressions—they represent a program terminated by a failed run-time check. A program that tries to access $\mathcal{H}n$ but $\mathcal{H}$ is undefined at n will be stuck, and violates spatial safety.

operational rules perform no bounds and null checks, which are instead inserted during compilation. Our compilation algorithm is evidence that CoreCHKCBox's semantics, despite its apparent use of fat pointers, faithfully represents Checked C's intended behavior. The algorithm also sheds light on how compilation can be implemented in the real Checked C compiler, while eschewing many important details (CoreC has many differences with LLVM IR).

Compilation is defined by extending CoreCHKCBox's typing judgment thusly:

$$\Gamma; \Theta; \rho \vdash_m e \gg \dot{e} : \tau$$

There is now a CoreC output $\dot{e}$ and an input ρ, which maps each `nt_array_ptr` variable `p` to a pair of *shadow variables* that keep `p`'s up-to-date upper and lower bounds; these may differ from the bounds in `p`'s type due to bounds widening.[5]

We formalize rules for this judgment in PLT Redex [10], following and extending our Coq development for CoreCHKCBox. To give confidence that compilation is correct, we use Redex's property-based random testing support to show that compiled-to $\dot{e}$ simulates e, for all e.

4.1. Approach

Due to space constraints, we explain the rules for compilation by example, using a C-like syntax; the complete rules are given in Appendix G. Each rule performs up to three tasks: (a) conversion of e to A-normal form; (b) insertion of dynamic checks; and (c) insertion of bounds widening expressions. A-normal form conversion is straightforward: compound expressions are handled by storing results of subexpressions into temporary variables, as in the following example.

```
let y=(x+1)+(6+1);   →   let a=x+1;
                         let b=6+1;
                         let y=a+b
```

This simplifies the management of effects from subexpressions. The next two steps of compilation are more interesting.

During compilation, Γ tracks the lower and upper bound associated with every pointer variable according to its type. At each declaration of a `nt_array_ptr` variable `p`, the compiler allocates two *shadow variables*, stored in $\rho(p)$; these are initialized to `p`'s declared bounds and will be updated during bounds widening.[6] Fig. 12 shows how an invocation of `strlen` on a null-terminated string is compiled into C code. Each dereference of a checked pointer requires a null check (See S-DefNull in Fig. 8), which the compiler makes explicit: Line 3 of the generated code has the null check on pointer `p` due to the `strlen`, and a similar check happens at line 8 due to the pointer arithmetic on `p`. Dereferences also require bounds checks: line 2 checks

5. Since lower bounds are never widened, the lower-bound shadow variable is unnecessary; we include it for uniformity.

6. Shadow variables are not used for `array_ptr` types (the bounds expressions are) since they are not subject to bounds widening.

```
1  /* p : ptr^c [(0,0) int]_nt */
2  /* ρ(p) = p_lo,p_hi */
3  {
4    let x = strlen(p);
5    if (x > 1) putchar(*(p+1));
6  }
```

→

```
1  {
2    assert(p_lo ≤ 0 && 0 ≤ p_hi); // bounds check
3    assert(p != 0); // null check
4    let x = strlen(p);
5    let p_hi_new = x;
6    p_hi = max(p_hi, p_hi_new);
7    if (x > 1) {
8      assert(p != 0); // null check for p + 1
9      let p_1 = p + 1;
10     assert(p_lo ≤ 1 && // bounds check for p + 1
11      1 ≤ p_hi);
12     putchar(*p_1);
13   }
14 }
```

Figure 12: Compilation Example for Check Insertions

`p` is in bounds before computing `strlen(p)`, while line 10 does likewise before computing `*(p+1)`.

For `strlen(p)` and conditionals `if(*p)`, the CoreCHKCBox semantics allows the upper bound of `p` to be extended. The compiler explicitly inserts statements to do so on `p`'s shadow bound variables. For example, Fig. 12 line 6 widens `p`'s upper bound if `strlen`'s result is larger than the existing bound. Lines 7–12 of the generated code in Fig. 13 show how bounds are widened when compiling expression `if(*p)`. If we find that the current `p`'s relative upper bound is equal to 0 (line 10), and `p`'s content is not null (line 8), we then increase the upper bound by 1 (line 11).

Fig. 13 also shows a dependent function call. Notice that the bounds for the array pointer `p` are not passed as arguments. Instead, they are initialized according to `p`'s type—see line 3 of the original CoreCHKCBox program at the top of the figure. Line 2 of the generated code sets the lower bound to 0 and line 3 sets the upper bound to `n`.

4.2. Comparison with Checked C Specification

The use of shadow variables for bounds widening is a key novelty of our compilation approach, and adds more precision to bounds checking at runtime compared to the official specification and current implementation of Checked C [39, 5.1.2, pg 85]. For example, the `safe_strcat` example of Fig. 9 compiles with the current Clang Checked C compiler but will fail with a runtime error. The statement `int x = strlen(dst)` at line 4 changes the statically determined upper bound of `dst` to `x`, which can be smaller than `n`, the full capacity of `dst`. The attempt to recover the full capacity of `dst` through a dynamic cast at line

```
int deref_array(n : int,
     p : ptr^c [(0,n) int]_nt) {
  /* ρ(p) = p_lo,p_hi */
  if (* p)
    (* (p + 1))
    else 0
}
...
/* p0 : ptr^c [(0,5) int]_nt */
deref_array(5, p0);
```

```
deref_array(n, p) {
  let p_lo = 0;
  let p_hi = n;
  /* runtime checks */
  assert(p_lo ≤0 && 0 ≤p_hi);
  assert(p != 0);
  let p_derefed = *p;
  if (p_derefed != 0) {
    /* widening */
    if (p_hi == 0) {
      p_hi = p_hi + 1;
    }
    /* null check before pointer arithmetic */
    assert(p != 0);
    let p0 = p + 1;
    assert(p_lo ≤1 && 1 ≤p_hi);
    (* p0)
  }
  else {
    0
  }
}
...
deref_array(5, p0);
```

Figure 13: Compilation Example for Dependent Functions

7 will always fail if the capacity `n` is checked against the statically determined new upper bound `x`. This problem can be worked around by invoking `strlen` on a temporary variable `tmp` instead of `dst` as in `safe_strcat_c` in Fig. 14 (lines 4-5). Likewise, if we were to add line `putchar(*(p+1));` after line 6 in the original code at the top of Fig. 12, the code will always fail: the Clang Checked C compiler (with the transliterated C code as its input) would check `p` against its *original* bounds `(0,0)` since the updated upper bound `x` is now out of the scope. Shadow variables address these problems because they retain widened bounds beyond the scope of variables that store them (i.e., `x` in both examples).

To make it match the specification, our compilation definition could easily eschew shadow variables and rely only on the type-based bounds expressions available in Γ for checking. However, doing so would force us to weaken the simulation theorem, reduce expressiveness, and/or force the semantics to be more awkward. We plan to work with the Checked C team to implement our approach in a future revision.

```
1  nt_array_ptr<char> safe_strcat_c
2     (nt_array_ptr<char> dst : count(n),
3      nt_array_ptr<char> src : count(0), int n) {
4    nt_array_ptr<char> tmp : count(n) = dst;
5    int x = strlen(tmp);
6    /* tmp now has x as its upper bound */
7    /* dst still has n as its upper bound */
8    int y = strlen(src);
9
10   if (x+y < n) {
11     for (int i = 0; i < y; ++i)
12       *(dst+x+i) = *(src+i);
13     *(dst+x+y) = '\0';
14     return dst;
15   }
16   return null;
17 }
```

Figure 14: Safe `strcat` in Checked C that avoids a run-time error exhibited by `safe_strcat` (Fig. 9) when compiled with the current Checked C compiler

4.3. Metatheory

We formalize both the compilation procedure and the simulation theorem in the PLT Redex model we developed for CoreChkCBox (see Sec. 3.1), and then attempt to falsify it via Redex's support for random testing. Redex allows us to specify compilation as logical rules (essentially, an extension of typing), but then execute it algorithmically to automatically test whether simulation holds. This process revealed several bugs in compilation and the theorem statement. We ultimately plan to prove simulation in the Coq model.

We use the notation $\gg$ to indicate the *erasure* of stack and heap—the rhs is the same as the lhs but with type annotations removed:

$$\mathcal{H} \gg \dot{\mathcal{H}}$$
$$\varphi \gg \dot{\varphi}$$

In addition, when $\Gamma; \emptyset \vdash \varphi$ and φ is well-formed, we write $(\varphi, \mathcal{H}, e) \gg (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$ to denote $\varphi \gg \dot{\varphi}$, $\mathcal{H} \gg \dot{\mathcal{H}}$ and $\Gamma; \Theta; \emptyset \vdash e \gg \dot{e} : \tau$ for some τ respectively. Γ is omitted from the notation since the well-formedness of φ and its consistency with respect to Γ imply that e must be closed under φ, allowing us to recover Γ from φ. Finally, we use $\dot{\rightarrow}^*$ to denote the transitive closure of the reduction relation of CoreC. Unlike the CoreChkCBox, the semantics of CoreC does not distinguish checked and unchecked regions.

Fig. 15 gives an overview of the simulation theorem.[7] The simulation theorem is specified in a way that is similar to the one by Merigoux et al. [25]. An ordinary simulation

7. We ellide the possibility of $\dot{e}_1$ evaluating to `bounds` or `null` in the diagram for readability.

property would replace the middle and bottom parts of the figure with the following:

$$(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1)$$

Instead, we relate two erased configurations using the relation $\sim$, which only requires that the two configurations will eventually reduce to the same state. We formulate our simulation theorem differently because the standard simulation theorem imposes a very strong syntactic restriction to the compilation strategy. Very often, $(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0)$ reduces to a term that is semantically equivalent to $(\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1)$, but we are unable to syntactically equate the two configurations due to the extra binders generated for dynamic checks and ANF transformation. In earlier versions of the Redex model, we attempted to change the compilation rules so the configurations could match syntactically. However, the approach scaled poorly as we added additional rules. This slight relaxation on the equivalence relation between target configurations allows us to specify compilation more naturally without having to worry about syntactic constraints.

Theorem 4 (Simulation ($\sim$)). For CORECHKCBOX expressions e_0, stacks φ_0, φ_1, and heap snapshots $\mathcal{H}_0$, $\mathcal{H}_1$, if $\mathcal{H}_0 \vdash \varphi_0$, $(\varphi_0, \mathcal{H}_0, e_0) \gg (\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0)$, and if there exists some r_1 such that $(\varphi_0, \mathcal{H}_0, e_0) \rightarrow_{\mathtt{c}} (\varphi_1, \mathcal{H}_1, r_1)$, then the following facts hold:

- if there exists e_1 such that $r = e_1$ and $(\varphi_1, \mathcal{H}_1, e_1) \gg (\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1)$, then there exists some $\dot{\varphi}$,$\dot{\mathcal{H}}$, $\dot{e}$, such that $(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$ and $(\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1) \dot{\rightarrow}^* (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$.
- if $r_1 = \mathtt{bounds}$ or $\mathtt{null}$, then we have $(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}_1, \dot{\mathcal{H}}_1, r_1)$ where $\varphi_1 \gg \dot{\varphi}_1$, $\mathcal{H}_1 \gg \dot{\mathcal{H}}_1$.

Our random generator (discussed in the next section) never produces `unchecked` expressions (whose behavior could be undefined), so we can only test a the simulation theorem as it applies to checked code. This limitation makes it unnecessary to state the other direction of the simulation theorem where e_0 is stuck, because Theorem 1 guarantees that e_0 will never enter a stuck state if it is well-typed in checked mode.

The current version of the Redex model has been tested against 20000 expressions with depth less than 10. Each expression can reduce multiple steps, and we test simulation between every two adjacent steps to cover a wider range of programs, particularly the ones that have a non-empty heap.

5. Random Testing via the Implementation

In addition to using the CORECHKCBOX Redex model to establish simulation of compilation (Section 4.3), we also used it to gain confidence that our model matches the Clang Checked C implementation. Disagreement on outcomes signals a bug in either the model or the compiler itself. Doing so allowed us to quickly iterate on the design of the model while adding new features, and revealed several bugs in the Clang Checked C implementation.

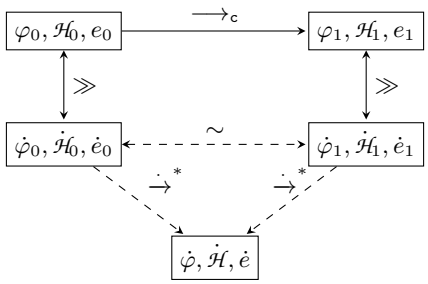

Figure 15: Simulation between CORECHKCBOX and COREC

Generating Well Typed Terms. For this random generation, we follow the approach of Pałka et al. [32] to generate well-typed Checked C terms by viewing the typing rules as generation rules. Suppose we have a context Γ, a mode m and a type τ, and we are trying to generate a well-typed expression. We can do that by reversing the process of type checking, selecting a typing rule and building up an expression in a way that satisfies the rule's premises.

Recall the typing rule for dereferencing an array pointer, which we depict below as G-DEFARR[8], color-coded to represent inputs and outputs of the generation process:[9]

G-DEFARR

$$\frac{\Gamma; \Theta \vdash_m e : \mathtt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad m \leq m'}{\Gamma; \Theta \vdash_m *e : \tau}$$

If we selected G-DEFARR for generating an expression, the generated expression has to have the form $*e$, for some e, to be generated according to the rule's premises. To satisfy the premise $\Gamma; \Theta \vdash_m e : \mathtt{ptr}^{m'}\ [\beta\ \tau]_\kappa$, we essentially need to make a recursive call to the generator, with appropriately adjusted inputs. However, the type in this judgment is not fixed yet—it contains three unknown variables: m', β, and κ—that need to be generated before making the call. Looking at the second premise informs that generation: if the input mode m is `u`, then m' needs to be `u` as well; if not, it is unconstrained, just like β and κ, and therefore all three are free to be generated at random. Thus, the recursive call to generate e can now be made, and the G-DEFARR rule returns $*e$ as its output.

Using such generator rules, we can create a generator for random well-typed terms of a given type in a straightforward manner: find all rules whose conclusion matches the given type and then randomly choose a candidate rule to perform the generation. To ensure that this process terminates, we follow the standard practice of using "fuel" to bound the depth of the generated terms; once the fuel is exhausted, only rules without recursive premises are selected [16].

8. Generator rules G-* correspond one to one with the type rules T-* in Sec. 3.3.

9. This input-output marking is commonly called a mode in the literature, but we eschew this term to avoid confusion with our pointer mode annotation.

Similar methods were used for generating top level functions and `struct` definitions.

While using just the typing-turned-generation rules is in theory enough to generate all well-typed terms, it's more effective in practice to try and exercise interesting patterns. As in Pałka et al. [32] this can be viewed as a way of adding admissible but redundant typing rules, with the sole purpose of using them for generation. For example, below is one such rule, G-ASTR, which creates an initialized null-terminated string that is statically cast into an array with bounds $(0, 0)$.

$$\text{G-ASTR}\quad \frac{\begin{array}{c} i \in \mathbb{N}^* \qquad n_0, \ldots, n_{i-1} \in \mathbb{Z} \qquad \texttt{fresh}(x) \\ \Gamma \vdash_m e' : \texttt{ptr}^c\ [(0, i)\ \texttt{int}]_{nt} \\ e = \texttt{let}\ x = e'\ \texttt{in}\ (\texttt{init}\ x\ \texttt{with}\ n_0, \ldots n_{i-1}); x \end{array}}{\Gamma \vdash_m (\texttt{ptr}^c\ [(0,0)\ \texttt{int}]_{nt})e : \texttt{ptr}^c\ [(0,0)\ \texttt{int}]_{nt}}$$

Given some positive number i, numbers $n_0, \ldots, n_{i-1}$, and a fresh variable x (which are arbitrarily generated), we can recursively generate a pointer e' with bounds $(0, i)$, and initialize it with the generated n_j using x to temporarily store the pointer.

This rule is particularly useful when combined with G-IFNT since there is a much higher chance of obtaining a non-zero value when evaluating $*p$ in the guard of `if`, skewing the distribution towards programs that enter the `then` branch. Relying solely on the type-based rules, entering the `then` branch requires that G-ASSIGNARR was chosen before G-IFNT, and that assignment would have to appear before `if`, which means additional G-LET rules would need to be chosen: this combination would therefore be essentially impossible to generate in isolation.

Adding admissible generation rules like G-ASTR in this manner, as described in Pałka et al. [32], is a manual process. It is guided by gathering statistics on the generated data and focusing on language constructs that appear underrepresented in the posterior distribution. For example, we arrived at the G-ASTR rule by recognizing that the pure type-based generation was not generating non-trivial null-terminated strings, and then analyzing the sequence of random choices that could lead to their generation.

Generating Ill-typed Terms. We can use generated well-typed terms to test our simulation theorem (Section 4) and test that CORECHKCBOX and Checked C Clang agree on what is type-correct. But it is also useful to generate ill-typed terms to test that CORECHKCBOX and Checked C Clang also agree on what is not. However, while it is easy to generate arbitrary ill-typed terms, they would be very unlikely to trigger any inconsistencies; those are far more likely to exist on the boundary between well- and ill-typedness. Therefore, we also manually added variations of existing generation rules modified to be slightly more permissive, e.g., by relaxing a single premise, thus allowing terms that are "a little" ill-typed to be generated. Unlike coming up with admissible generation rules like G-ASTR (which is quite challenging to automate), systematically and automatically relaxing premises of existing rules seems feasible, and worthwhile future work.

Random Testing for Language Design. We used our Redex model and random generator to successfully guide the design of our formal model, and indeed the Clang Checked C implementation itself, which is being actively developed. To that end, we implemented a conversion tool that converts CORECHKCBOX into a subset of the Checked C language and ensured that model and implementation exhibit the same behavior (accept and reject the same programs and yield the same return value).

This approach constitutes an interesting twist to traditional model-based checking approaches. Usually, one checks that the implementation and model agree on all inputs *of the implementation*, with the goal of covering as many behaviors as possible. This is the case, for example, in Guha et al. [12], where they use real test suites to demonstrate the faithfullness of their core calculus to Javascript. Our approach and goal in this work is essentially the opposite: as the Clang Checked C implementation does not fully implement the Checked C spec, there is little hope of covering all terms that are generated by Clang Checked C. Instead, we're looking for *inconsistencies*, which could be caused by bugs either in the Clang Checked C compiler or our own model.

One inconsistency we found comes from the following:

```
1  array_ptr<char> fun(void) : count(3) {
2    array_ptr<char> x : count(3);
3    x = calloc(3, sizeof(char));
4    return x+3;
5  }
6  int main(void) {
7    *(fun()) = 0;
8    return 0;
9  }
```

In this code, the function `fun` is supposed to return a checked array pointer of size 3. Internally, it allocates such an array, but instead of returning the pointer `x` to that array, it increments that pointer by 3. Then, the `main` function just calls `fun`, and tries to assign 0 to its result. Our model correctly rules out this program, while the Clang Checked C implementation happily accepted this out-of-bounds assignment. Interestingly, it correctly rejected programs where the array had size 1 or 2. This inconsistency has been fixed in the latest version of the compiler.

We also found the opposite kind of inconsistency—programs that the Clang Checked C implementation rejects contrary to the spec. For instance:[10]

```
1  array_ptr<int> f(void) : count(5) {
2    array_ptr<int> x : count(5) =
3      calloc<int>(5, sizeof(int));
4    return x;
5  }
6  array_ptr<int> g(void ) : count(5) {
7    array_ptr<int> x : count(5) =
8      calloc<int>(5, sizeof(int));
9    return x+3;
10 }
```

10. After minimization, this turned out to be a known issue: https://github.com/microsoft/checkedc-clang/issues/1008

```
11 int main(void) {
12   return *(0 ? g() : f() + 3);
13 }
```

In this piece of code both `f` and `g` functions compute a pointer to the same index in an array of size 5 (as `f` calls `g`). The `main` function then creates a ternary expression whose branches call `f` and `g`, but the Clang Checked C implementation rejects this program, as its static analysis is not sophisticated enough to detect that both branches have the same type.

6. Related Work

Our work is most closely related to prior formalizations of C(-like) languages that aim to enforce memory safety, but it also touches on C-language formalization in general.

Formalizing C and Low-level code. A number of prior works have looked at formalizing the semantics of C, including CompCert [1, 19], Ellison and Rosu [8], Kang et al. [15], and Memarian et al. [22, 23]. These works also model pointers as logically coupled with either the bounds of the blocks they point to, or provenance information from which bounds can be derived. None of these is directly concerned with enforcing spatial safety, and that is reflected in the design. For example, memory itself is not be represented as a flat address space, as in our model or real machines, so memory corruption due to spatial safety violations, which Checked C's type system aims to prevent, may not be expressible. That said, these formalizations consider much more of the C language than does CoreChkCBox, since they are interested in the entire language's behavior.

Spatially Safe C Formalizations. Several prior works formalize C-language transformations or C-language dialects aiming to ensure spatial safety. Hathhorn et al. [13] extends the formalization of Ellison and Rosu [8] to produce a semantics that detects violations of spatial safety (and other forms of undefinedness). It uses a CompCert-style memory model, but "fattens" logical pointer representations to facilitate adding side conditions similar to CoreChkCBox's. Its concern is bug finding, not compiling programs to use this semantics.

CCured [30] and Softbound [28] implement spatially safe semantics for normal C via program transformation. Like CoreChkCBox, both systems' operational semantics annotate pointers with their bounds. CCured's equivalent of array pointers are compiled to be "fat," while SoftBound compiles bounds metadata to a separate hashtable, thus retaining binary compatibility at higher checking cost. Checked C uses static type information to enable bounds checks without need of pointer-attached metadata, as we show in Section 4. Neither CCured nor Softbound models null-terminated array pointers, whereas our semantics ensures that such pointers respect the zero-termination invariant, leveraging bounds widening to enhance expressiveness.

Cyclone [11, 14] is a C dialect that aims to ensure memory safety; its pointer types are similar to CCured. Cyclone's formalization [11] focuses on the use of *regions* to ensure temporal safety; it does not formalize arrays or threats to spatial safety. Deputy [4, 45] is another safe-C dialect that aims to avoid fat pointers; it was an initial inspiration for Checked C's design [7], though it provides no specific modeling for null-terminated array pointers. Deputy's formalization [4] defines its semantics directly in terms of compilation, similar in style to what we present in Section 4. Doing so tightly couples typing, compilation, and semantics, which are treated independently in CoreChkCBox. Separating semantics from compilation isolates meaning from mechanism, easing understandability. Indeed, it was this separation that led us to notice the limitation with Checked C's handling of bounds widening.

The most closely related work is the formalization of Checked C done by Ruef et al. [34]. They present the type system and semantics of a core model of Checked C, mechanized in Coq, and were the first to prove a blame theorem. CoreChkCBox's Coq-based development (Section 3) substantially extends theirs to include conditionals, dynamically bounded array pointers with dependent types, null-terminated array pointers, dependently typed functions, and subtyping. They postulate that pointer metadata can be erased in a real implementation, but do not show it. Our CoreChkCBox compiler, formalized and validated in PLT Redex via randomized testing, demonstrates that such metadata *can* be erased; we found that erasure was non-obvious once null-terminated pointers and bounds widening were considered.

7. Conclusion and Future Work

This paper presented CoreChkCBox, a formalization of an extended core of the Checked C language which aims to provide spatial memory safety. CoreChkCBox models dynamically sized and null-terminated arrays with dependently typed bounds that can additionally be widened at runtime. We prove, in Coq, the key safety property of Checked C for our formalization, *blame*: if a mix of checked and unchecked code gives rise to a spatial memory safety violation, then this violation originated in an unchecked part of the code. We also show how programs written in CoreChkCBox (whose semantics leverage fat pointers) can be compiled to CoreC (which does not) while preserving their behavior. We developed a version of CoreChkCBox written in PLT Redex, and used a custom term generator in conjunction with Redex's randomized testing framework to give confidence that compilation is correct. We also used this framework to cross-check CoreChkCBox against the Checked C compiler, finding multiple inconsistencies in the process.

As future work, we wish to extend CoreChkCBox to model more of Checked C, with our Redex-based testing framework guiding the process. The most interesting Checked C feature not yet modeled is *interop types* (itypes), which are used to simplify interactions with unchecked code via function calls. A function whose parameters are itypes can be passed checked or unchecked pointers depending on whether the caller is in a checked region. This feature allows

for a more modular C-to-Checked C porting process, but complicates reasoning about blame. A more ambitious next step would be to extend an existing formally verified framework for C, such as CompCert [18] or VeLLVM [44], with Checked C features, towards producing a verified-correct Checked C compiler. We believe that CoreChkCBox's Coq and Redex models lay the foundation for such a step, but substantial engineering work remains.

References

[1] Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. ISSN 1573-0670. doi: 10.1007/s10817-009-9148-3. URL http://dx.doi.org/10.1007/s10817-009-9148-3.

[2] BlueHat. Memory corruption is still the most prevalent security vulnerability. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019. Accessed: 2020-02-11.

[3] c2rust. C to rust translation, refactoring, and cross-checking. https://c2rust.com/, 2018.

[4] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-Level Programming. In *Proceedings of European Symposium on Programming (ESOP '07)*, 2007.

[5] Junhan Duan, Yudi Yang, Jie Zhou, and John Criswell. Refactoring the FreeBSD kernel with Checked C. In *Proceedings of the 2020 IEEE Cybersecurity Development Conference (SecDev)*, 2020.

[6] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[7] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018. doi: 10.1109/SecDev.2018.00015.

[8] Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103719. URL http://doi.acm.org/10.1145/2103656.2103719.

[9] Mehmet Emre, Kyle Dewey, Ryan Schroeder, and Ben Hardekopf. Translating C to safer Rust. *In Proceedings of the 2021 ACM on Programming Languges (PACMPL)*, 5(OOPSLA), 2021.

[10] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. ISBN 0262062755.

[11] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *PLDI*, 2002.

[12] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of Javascript. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, page 126–150, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642141064.

[13] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. *SIGPLAN Not.*, 50(6):336–345, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737979. URL https://doi.org/10.1145/2813885.2737979.

[14] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, , and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, 2002. USENIX.

[15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-pointer Casts. *SIGPLAN Not.*, 50(6):326–335, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2738005. URL http://doi.acm.org/10.1145/2813885.2738005.

[16] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, August 2018. Version 1.0. http://www.cis.upenn.edu/~bcpierce/sf.

[17] Per Larson. Migrating legacy code to Rust. RustConf 2018 talk, August 2018.

[18] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10/c9sb7q. URL http://doi.acm.org/10.1145/1538788.1538814.

[19] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL https://hal.inria.fr/hal-00703441.

[20] Liyi Li, Yiyun Liu, Deena L. Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. A formal model of Checked C. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, August 2022.

[21] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to checked c by 3c. *Proceedings of the ACM on Programming Languages*, 6 (OOPSLA1):1–29, 2022.

[22] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the Depths of C: Elaborating the de Facto Standards. *SIGPLAN Not.*, 51(6):1–15, June 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908081. URL https://doi.org/10.1145/2980983.2908081.

[23] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, January 2019. ISSN 2475-1421. doi: 10.1145/3290380. URL http://doi.acm.org/10.1145/3290380.

[24] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. 2022.

[25] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A Programming Language for the Law. *arXiv preprint arXiv:2103.03198*, 2021.

[26] Mozilla. Rust Programming Language. https://www.rust-lang.org/, 2021.

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44 (6):245–258, 2009.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542504. URL https://doi.org/10.1145/1542476.1542504.

[29] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[30] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), 2005.

[31] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[32] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0592-1. doi: 10.1145/1982595.1982615. URL http://doi.acm.org/10.1145/1982595.1982615.

[33] Ricardo Peña. An Introduction to Liquid Haskell. *Electronic Proceedings in Theoretical Computer Science*, 237:68–80, Jan 2017. ISSN 2075-2180. doi: 10.4204/eptcs.237.5. URL http://dx.doi.org/10.4204/EPTCS.237.5.

[34] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving Safety Incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17138-4.

[35] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with checked c. In *International Conference on Principles of Security and Trust*, pages 76–98. Springer, Cham, 2019.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[37] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS*. ACM, 2004.

[38] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[39] David Tarditi. Extending C with Bounds Safety and Improved Type Safety, 2021. URL https://github.com/secure-sw-dev/checkedc/.

[40] David Tarditi, Archibald Samuel Elliott, Andrew Ruef, and Michael Hicks. Checked C: Making C safe by extension. In *IEEE Cybersecurity Development Conference 2018 (SecDev)*, September 2018.

[41] CVE Trends. Cve trends. https://www.cvedetails.com/vulnerabilities-by-types.php, 2021. Accessed: 2020-10-11.

[42] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. *SIGPLAN Not.*, 49(9):269–282, August 2014. ISSN 0362-1340. doi: 10.1145/2692915.2628161. URL https://doi.org/10.1145/2692915.2628161.

[43] Anna Zeng and Will Crichton. Identifying barriers to adoption for rust through online discourse. *arXiv preprint arXiv:1901.01001*, 2019.

[44] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103709. URL http://doi.acm.org/10.1145/2103621.2103709.

[45] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating System Design and Implementation (OSDI'06)*, Seattle, Washington, 2006. USENIX Association.

[46] Jie Zhou. The benefits and costs of using fat pointers for temporal memory safety. Poster presentation at PLDI 2021 student research competition (silver medalist), June 2021.

Appendix

1. Differences with the Coq and Redex Models

The Coq and Redex models of CORECHKCBOX may be found at https://github.com/plum-umd/checkedc. The Coq model's syntax is slightly different from that in Fig. 4. In particular, the arguments in a function call are restricted to variables and constants, according to a separate well-formedness condition. A function call `f(e)` can always be written in `let x = e in f(x)` to cope. In addition, conditionals have two syntactic forms: `EIf` is a normal conditional, and `EIfDef` is one whose boolean guard is of the form $*x$. By syntactically distinguishing these two cases, the Coq model does not need the *[prefer]* rule for `if (*x)`... forms as in Fig. 6. The Redex model *does* prioritize such forms but not the same way as in the figure. It uses a variation of the S-VAR rule: The modified rule is equipped with a precondition that is false whenever S-IFNTT is applicable.

The Coq model uses a runtime stack φ as described at the start of Sec. 3.2. The Redex model introduces let bindings during evaluation to simulate a runtime stack. For example, consider the expression $e \equiv$ `let` $x = (5 :$ `int`$)$ `in` $x + x$. Expression e first steps to `let` $x = (5 :$ `int`$)$ `in` $(5{:}\texttt{int}) + x$, which in turns steps to `let` $x = (5 :$ `int`$)$ `in` $(5{:}\texttt{int}) + (5{:}\texttt{int})$. Since the rhs of x is a value, the let binding in e effectively functions as a stack that maps from x to $5 : \texttt{int}$. The let form remains in the expression and lazily replaces the variables in its body. The let form can be removed from the expression only if its body is evaluated to a value, e.g., `let` $x = (5 : \texttt{int})$ `in` $(10{:}\texttt{int})$ steps to $10 : \texttt{int}$. The rule for popping let bindings in this manner corresponds to the S-RET rule in Fig. 8. Leveraging let bindings adds complexity to the semantics but simplifies typing/consistency and term generation during randomized testing.

2. Typing Rules for Literal Pointers

The typing of integer literals, which can also be pointers to the heap, was presented in Sec. 3.4 in Fig. 7. Here we describe these rules further.

The variable type rule (T-VAR) simply checks if a given variable has the defined type in Γ; the constant rule (T-CONST) is slightly more involved. First, it ensures that the type annotation τ does not contain any free variables. More importantly, it ensures that the literal itself is well typed using an auxilliary typing relation $\mathcal{H}; \sigma \vdash n : \tau$.

If the literal's type is an integer, an unchecked pointer, or a null pointer, it is well typed, as shown by the top three rules in Fig. 7. However, if it is a checked pointer $\texttt{ptr}^{\texttt{c}}\ \omega$, we need to ensure that what it points to in the heap is of the appropriate pointed-to type (ω), and also recursively ensure that any literal pointers reachable this way are also well-typed. This is captured by the bottom rule in the figure, which states that for every location $n + i$ in the pointers' range $[n, n + \mathit{size}(\omega))$, where *size* yields the size of its argument, then the value at the location $\mathcal{H}(n + i)$ is also well-typed. However, as heap snapshots can contain cyclic structures (which would lead to infinite typing deriviations), we use a scope σ to assume that the original pointer is well-typed when checking the types of what it points to. The middle rule then accesses the scope to tie the knot and keep the derivation finite, just like in Ruef et al. [34].

3. Other Semantic Rules

Fig. 16 shows the remaining semantic rules for CORECHKCBOX. We explain a selected few rules in this subsection.

Rule S-VAR loads the value for x in stack φ. Rule S-DEFARRAY dereferences an array pointer, which is similar to the Rule S-DEFNTARRAY in Fig. 8 (dealing with null-terminated array pointers). The only difference is that the range of 0 is at $[n_l, n_h)$ not $[n_l, n_h]$, meaning that one cannot dereference the upper-bound position in an array. Rules DEFARRAYBOUND and DEFNTARRAYBOUND describe an error case for a dereference operation. If we are dereferencing an array/NT-array pointer and the mode is `c`, 0 must be in the range from n_l to n_h (meaning that the dereference is in-bound); if not, the system results in a `bounds` error. Obviously, the dereference of an array/NT-array pointer also experiences a `null` state transition if $n \leq 0$.

Rules S-MALLOC and S-MALLOCBOUND describe the `malloc` semantics. Given a valid type ω_a that contains no free variables, `alloc` function returns an address pointing at the first position of an allocated space whose size is equal to the size of ω_a, and a new heap snapshot $\mathcal{H}'$ that marks the allocated space for the new allocation. The `malloc` is transitioned to the address n with the type $\texttt{ptr}^{\texttt{c}}\ \omega_a$ and new updated heap. It is possible for `malloc` to transition to a `bounds` error if the ω_a is an array/NT-array type $[(n_l, n_h)\ \tau]_\kappa$, and either $n_l \neq 0$ or $n_h \leq 0$. This can happen when the bound variable is evaluated to a bound constant that is not desired.

4. Subtyping for dependent types

The subtyping relation given in Fig. 11 involves dependent bounds, i.e., bounds that may refer to variables. To decide premises $b \leq b'$, we need a decision procedure that accounts for the possible values of these variables. This process considers Θ, tracked by the typing judgment, and φ, the current stack snapshot (when performing subtyping as part of the type preservation proof).

Definition 7 (Inequality).

- $n \leq m$ if n is less than or equal to m.
- $x + n \leq x + m$ if n is less than or equal to m.
- All other cases result in `false`.

To capture bound variables in dependent types, the Checked C subtyping relation ($\sqsubseteq$) is parameterized by a restricted stack snapshot $\varphi|_\rho$ and the predicate map Θ, where φ is a stack and ρ is a set of variables. $\varphi|_\rho$ means to restrict the domain of φ to the variable set ρ. Clearly, we have the

S-VAR
$$(\varphi, \mathcal{H}, x) \longrightarrow (\varphi, \mathcal{H}, \varphi(x))$$

S-DEFARRAY
$$\frac{\mathcal{H}(n) = n_a : \tau_a \qquad 0 \in [n_l, n_h)}{(\varphi, \mathcal{H}, *n : \texttt{ptr}^{\texttt{c}}\ [(n_l, n_h)\ \tau]_{nt}) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}$$

S-DEFARRAYBOUND
$$\frac{0 \notin [n_l, n_h)}{(\varphi, \mathcal{H}, *n : \texttt{ptr}^{\texttt{c}}\ [(n_l, n_h)\ \tau]_{\kappa}) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}$$

S-DEFNTARRAYBOUND
$$\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, *n : \texttt{ptr}^{\texttt{c}}\ [(n_l, n_h)\ \tau]_{nt}) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}$$

S-ASSIGN
$$\frac{\mathcal{H}(n) = n_a : \tau_a}{(\varphi, \mathcal{H}, *n : \texttt{ptr}^{\texttt{c}}\ \tau = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}[n \mapsto n_1 : \tau], n_1 : \tau)}$$

S-ASSIGNNULL
$$(\varphi, \mathcal{H}, *0 : \texttt{ptr}^{\texttt{c}}\ \omega = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})$$

S-ASSIGNARRBOUND
$$\frac{0 \notin [n_l, n_h)}{(\varphi, \mathcal{H}, *n : \texttt{ptr}^{\texttt{c}}\ [(n_l, n_h)\ \tau]_{\kappa} = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}$$

S-MALLOC
$$\frac{\varphi(\omega) = \omega_a \qquad \texttt{alloc}(\mathcal{H}, \omega_a) = (n, \mathcal{H}')}{(\varphi, \mathcal{H}, \texttt{malloc}(\omega,)) \longrightarrow (\varphi, \mathcal{H}', n : \texttt{ptr}^{\texttt{c}}\ \omega_a)}$$

S-MALLOCBOUND
$$\frac{\varphi(\omega) = [(n_l, n_h)\ \tau]_{\kappa} \qquad (n_l \neq 0 \vee n_h \leq 0)}{(\varphi, \mathcal{H}, \texttt{malloc}(\omega,)) \longrightarrow (\varphi, \mathcal{H}', \texttt{bounds})}$$

S-IFT
$$\frac{n \neq 0}{(\varphi, \mathcal{H}, \texttt{if}\ (n : \tau)\ e_1\ \texttt{else}\ e_2) \longrightarrow (\varphi, \mathcal{H}, e_1)}$$

S-IFF
$$(\varphi, \mathcal{H}, \texttt{if}\ (0 : \tau)\ e_1\ \texttt{else}\ e_2) \longrightarrow (\varphi, \mathcal{H}, e_2)$$

S-UNCHECKED
$$(\varphi, \mathcal{H}, \texttt{unchecked}(n : \tau)\{\longrightarrow\}) (\varphi, \mathcal{H}, n : \tau)$$

S-STR
$$\frac{0 \in [n_l, n_h] \qquad n_a \leq n_h \qquad \mathcal{H}(n + n_a) = 0 \qquad (\forall i. n \leq i < n + n_a \Rightarrow (\exists n_i\ t_i. \mathcal{H}(n + i) = n_i : \tau_i \wedge n_i \neq 0))}{(\varphi, \mathcal{H}, \texttt{strlen}(n : \texttt{ptr}^{m}\ [(n_l, n_h)\ \tau])) \longrightarrow (\varphi, \mathcal{H}, n_a : \texttt{int})}$$

S-STRBOUNDS
$$\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, \texttt{strlen}(n : \texttt{ptr}^{c}\ [(n_l, n_h)\ \tau])) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}$$

S-STRNULL
$$(\varphi, \mathcal{H}, \texttt{strlen}(0 : \texttt{ptr}^{c}\ [(n_l, n_h)\ \tau])) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})$$

S-ADD
$$\frac{n = n_1 + n_2}{(\varphi, \mathcal{H}, n_1 : \texttt{int} + n_2 : \texttt{int}) \longrightarrow (\varphi, \mathcal{H}, n)}$$

S-ADDARR
$$\frac{n = n_1 + n_2 \qquad n_l' = n_l - n_2 \qquad n_h' = n_h - n_2}{(\varphi, \mathcal{H}, n_1 : \texttt{ptr}^{m}\ [(n_l, n_h)\ \tau]_{\kappa} + n_2 : \texttt{int}) \longrightarrow (\varphi, \mathcal{H}, n : \texttt{ptr}^{m}\ [(n_l', n_h')\ \tau]_{\kappa})}$$

S-ADDARRNULL
$$n(\varphi, \mathcal{H}, 0 : \texttt{ptr}^{c}\ [(n_l, n_h)\ \tau]_{\kappa} + n_2 : \texttt{int}) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})$$

Figure 16: Remaining CORECHKCBOX Semantics Rules (extends Fig. 8)

relation: $\varphi|_\rho \subseteq \varphi$. $\sqsubseteq$ being parameterized by $\varphi|_\rho$ refers to that when we compare two bounds $b \leq b'$, we actually do $\varphi|_\rho(b) \leq \varphi|_\rho(b')$ by interpreting the variables in b and b' with possible values in $\varphi|_\rho$. Let's define a subset relation $\preceq$ for two restricted stack snapshot $\varphi|_\rho$ and $\varphi'|_\rho$:

Definition 8 (Subset of Stack Snapshots). Given two $\varphi|_\rho$ and $\varphi'|_\rho$, $\varphi|_\rho \preceq \varphi'|_\rho$, iff for $x \in \rho$ and y, $(x, y) \in \varphi|_\rho \Rightarrow (x, y) \in \varphi'|_\rho$.

For every two restricted stack snapshots $\varphi|_\rho$ and $\varphi'|_\rho$, such that $\varphi|_\rho \preceq \varphi'|_\rho$, we have the following theorem in Checked C (proved in Coq):

Theorem 5 (Stack Snapshot Theorem). Given two types τ and τ', two restricted stack snapshots $\varphi|_\rho$ and $\varphi'|_\rho$, if $\varphi|_\rho \preceq \varphi'|_\rho$, and $\tau \sqsubseteq \tau'$ under the parameterization of $\varphi|_\rho$, then $\tau \sqsubseteq \tau'$ under the parameterization of $\varphi'|_\rho$.

Clearly, for every $\varphi|_\rho$, we have $\emptyset \preceq \varphi|_\rho$. The type checking stage is a compile-time process, so $\varphi|_\rho$ is $\emptyset$ at the type checking stage. Stack snapshots are needed for proving type preserving, as variables in bounds expressions are evaluated away.

As mentioned in the main text, $\sqsubseteq$ is also parameterized by Θ, which provides the range of allowed values for a bound variable; thus, more $\sqsubseteq$ relation is provable. For example, in Fig. 9, the `strlen` operation in line 4 turns the type of `dst` to be $\texttt{ptr}^{\texttt{c}}\ [(0, x)\ \texttt{int}]_{nt}$ and extends the upper bound to `x`. In the `strlen` type rule, it also inserts a predicate $\texttt{x} \geq 0$ in Θ; thus, the cast operation in line 16 is valid because $\texttt{ptr}^{\texttt{c}}\ [(0, x)\ \texttt{int}]_{nt} \sqsubseteq \texttt{ptr}^{\texttt{c}}\ [(0, 0)\ \texttt{int}]_{nt}$ is provable when we know $\texttt{x} \geq 0$.

Note that if φ and Θ are $\emptyset$, we do only the syntactic $\leq$ comparison; otherwise, we apply φ to both sides of $\sqsubseteq$, and then determine the $\leq$ comparasion based on a Boolean predicate decision procedure on top of Θ. This process allows us to type check both an input expression and the intermediate expression after evaluating an expression.

Struct Syntax:

Type	$\texttt{struct}\ T$
Structdefs	$D \in T \rightharpoonup fs$
Fields	$fs ::= \tau\ \texttt{f} \mid \tau\ \texttt{f}; fs$

Struct Subtype:

$$D(T) = fs \wedge fs(0) = \texttt{nat} \Rightarrow \texttt{ptr}^m\ \texttt{struct}\ T \sqsubseteq \texttt{ptr}^m\ \texttt{nat}$$

$$D(T) = fs \wedge fs(0) = \texttt{nat} \wedge 0 \leq b_l \wedge b_h \leq 1 \Rightarrow \texttt{ptr}^m\ \texttt{struct}\ T \sqsubseteq \texttt{ptr}^m\ [(b_l, b_h)\ \texttt{nat}]$$

Struct Type Rule:

T-Struct
$$\frac{\Gamma; \Theta \vdash_m e : \texttt{ptr}^m\ \texttt{struct}\ T \qquad D(T) = fs \qquad fs(f) = \tau_f}{\Gamma; \Theta \vdash_m \&e{\rightarrow}f : \texttt{ptr}^m\ \tau_f}$$

Struct Semantics:

S-StructChecked
$$\frac{n > 0 \qquad D(T) = fs \qquad fs(f) = \tau_a \qquad n_a = \texttt{index}(fs, f)}{(\varphi, \mathcal{H}, \&n{:}\texttt{ptr}^{\texttt{c}}\ \texttt{struct}\ T{\rightarrow}f) \longrightarrow (\varphi, \mathcal{H}, n_a{:}\texttt{ptr}^{\texttt{c}}\ \tau_a)}$$

S-StructNull
$$\frac{n = 0}{(\varphi, \mathcal{H}, \&n{:}\texttt{ptr}^{\texttt{c}}\ \texttt{struct}\ T{\rightarrow}f) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})}$$

S-StructUnChecked
$$\frac{D(T) = fs \qquad fs(f) = \tau_a \qquad n_a = \texttt{index}(fs, f)}{(\varphi, \mathcal{H}, \&n{:}\texttt{ptr}^{\texttt{u}}\ \texttt{struct}\ T{\rightarrow}f) \longrightarrow (\varphi, \mathcal{H}, n_a{:}\texttt{ptr}^{\texttt{u}}\ \tau_a)}$$

Figure 18: CoreChkCBox Struct Definitions

$$\Gamma \vdash n \qquad \frac{x : \texttt{int} \in \Gamma}{\Gamma \vdash x + n} \qquad \frac{\Gamma \vdash b_l \qquad \Gamma \vdash b_h}{\Gamma \vdash (b_l, b_h)} \qquad \Gamma \vdash \texttt{int}$$

$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \tau}{\Gamma \vdash \texttt{ptr}^m\ [\beta\ \tau]_\kappa} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \texttt{ptr}^m\ \tau} \qquad \frac{T \in D}{\Gamma \vdash \texttt{ptr}^m\ \texttt{struct}\ T}$$

Figure 19: Well-formedness for Types and Bounds

T-Def
$$\frac{\Gamma; \Theta \vdash_m e : \texttt{ptr}^{m'}\ \tau \qquad m \leq m'}{\Gamma; \Theta \vdash_m *e : \tau}$$

T-Mac
$$\Gamma; \Theta \vdash_m \texttt{malloc}(\omega, :)\texttt{ptr}^{\texttt{c}}\ \omega$$

T-Add
$$\frac{\Gamma; \Theta \vdash_m e_1 : \texttt{int} \qquad \Gamma; \Theta \vdash_m e_2 : \texttt{int}}{\Gamma; \Theta \vdash_m (e_1 + e_2) : \texttt{int}}$$

T-Ind
$$\frac{\Gamma; \Theta \vdash_m e_1 : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad \Gamma; \Theta \vdash_m e_2 : \texttt{int} \qquad m \leq m'}{\Gamma; \Theta \vdash_m *(e_1 + e_2) : \tau}$$

T-Assign
$$\frac{\Gamma; \Theta \vdash_m e_1 : \texttt{ptr}^{m'}\ \tau \qquad \Gamma; \Theta \vdash_m e_2 : \tau' \qquad \tau' \sqsubseteq \tau \qquad m \leq m'}{\Gamma; \Theta \vdash_m *e_1 = e_2 : \tau}$$

T-IndAssign
$$\frac{\Gamma; \Theta \vdash_m e_1 : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad \Gamma; \Theta \vdash_m e_2 : \texttt{int} \qquad \Gamma; \Theta \vdash_m e_3 : \tau' \qquad \tau' \sqsubseteq \tau \qquad m \leq m'}{\Gamma; \sigma \vdash_m *(e_1 + e_2) = e_3 : \tau}$$

5. Other Type Rules

Here we show the type rules for other Checked C operations in Fig. 17. Rule T-Def is for dereferencing a non-array pointer. The statement $m \leq m'$ ensures that no unchecked pointers are used in checked regions. Rule T-Mac deals with `malloc` operations. There is a well-formedness check to require that the possible bound variables in ω must be in the domain of Γ (see Fig. 19). This is similar to the well-formedness assumption of the type environment (Definition 1) Rule T-Add deals with binary operations whose subterms are integer expressions, while rule T-Ind serves the case for pointer arithmetic. For simplicity, in the Checked C formalization, we do not allow arbitrary pointer arithmetic. The only pointer arithmetic operations allowed are the forms shown in rules T-Ind and T-IndAssign in Fig. 17. Rule T-Assign assigns a value to a non-array pointer location. The predicate $\tau' \sqsubseteq \tau$ requires that the value being assigned is a subtype of the pointer type. The T-IndAssign rule is an extended assignment operation for handling assignments for array/NT-array pointers with pointer arithmetic. Rule T-Unchecked type checks `unchecked` blocks.

6. Struct Pointers

Checked C has `struct` types and `struct` pointers. Fig. 18 contains the syntax of `struct` types as well as new subtyping relations built on the `struct` values. For a `struct` typed value, Checked C has a special operation for it, which is $\&e{\rightarrow}f$. This operation indexes the f-th position `struct` T item, if the expression e is evaluated to a `struct` pointer $\texttt{ptr}^m$ `struct` T. Rule T-Struct in Fig. 18 describes its typing behavior. Rules S-StructChecked and S-StructUnChecked describe the semantic behaviors of $\&e{\rightarrow}f$ on a given `struct` `checked`/`unchecked` pointers, while rule S-StructNull describes a `checked` `struct` null-pointer case. In our Coq/Redex formalization, we include the `struct` values and the operation $\&e{\rightarrow}f$. We omit it in the main text due to the paper length limitation.

7. The Compilation Rules

Fig. 23 and Fig. 24 shows the syntax for CoreC, the target language for compilation. We syntactically restrict the expressions to be in A-normal form to simplify the presentation of the compilation rules. In the Redex model, we occasionally break this constraint to speed up the performance

$$\frac{\Gamma \vdash \overline{x} : \overline{\tau} \qquad \Gamma[\overline{x} \mapsto \overline{\tau}] \vdash \tau \qquad \Gamma[\overline{x} \mapsto \overline{\tau}]; \Theta \vdash_{\mathtt{c}} e : \tau}{\Gamma \vdash \tau\ (\overline{x} : \overline{\tau})\ e} \qquad \Gamma \vdash \cdot$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma[x \mapsto \tau] \vdash \overline{x} : \overline{\tau}}{\Gamma \vdash x : \tau, \overline{x} : \overline{\tau}}$$

Figure 20: Well-formedness for functions

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau\ \mathtt{f}} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash fs}{\Gamma \vdash \tau\ \mathtt{f}; fs}$$

Figure 21: Well-formedness for structs

$$\frac{\Gamma[\overline{x} \mapsto \overline{\tau}]; \emptyset \vdash e \gg \dot{e} : \tau}{\Gamma \vdash \tau\ (\overline{x} : \overline{\tau})\ e \gg (\overline{x})\ \dot{e}}$$

Figure 22: Compilation Rules for Functions

of random testing by removing unnecessary let bindings. To allow explicit runtime checks, we include `bounds` and `null` as part of CoreC expressions which, once evaluated, result in an corresponding error state. $x = \dot{a}$ is a new syntactic form that modifies the stack variable x with the result of $\dot{a}$. It is essential for bounds widening. $\leq$ and $-$ are introduced to operate on bounds and decide whether we need to halt with a bounds error or widen a null-terminated string.

Atoms	$\dot{a}$	::=	$n \mid x$
C-Expressions	$\dot{c}$	::=	$\dot{a} \mid \mathtt{strlen}(\dot{a}) \mid \mathtt{malloc}(\dot{a}, \mid) f(\overline{\dot{a}})$
		$\mid$	$\mid \dot{a} \circ \dot{a} \mid *\dot{a}$
		$\mid$	$*\dot{a} = \dot{a} \mid x = \dot{a} \mid \mathtt{if}\ (\dot{a})\ \dot{e}\ \mathtt{else}\ \dot{e}$
		$\mid$	$\mathtt{bounds} \mid \mathtt{null}$
Expressions	$\dot{e}$	::=	$\dot{c} \mid \mathtt{let}\ x = \dot{c}\ \mathtt{in}\ \dot{e}$
Binops	$\circ$	::=	$+ \mid - \mid \leq$
Closure	$\dot{C}$	::=	$\square \mid \mathtt{let}\ x = \dot{a}\ \mathtt{in}\ \dot{C}$
		$\mid$	$\mathtt{if}\ (\dot{a})\ \dot{e}\ \mathtt{else}\ \dot{C} \mid \mathtt{if}\ (\dot{a})\ \dot{C}\ \mathtt{else}\ \dot{e}$
Bounds Map	ρ	$\in$	$\mathtt{Var} \rightharpoonup \mathtt{Var} \times \mathtt{Var}$

Figure 23: CoreC Syntax

$\dot{\mu}$	::=	$n \mid \bot$
$\dot{c}$	::=	$\ldots \mid \mathtt{ret}(x, \dot{\mu}, \dot{e})$
$\dot{H}$	$\in$	$\mathbb{Z} \rightharpoonup \mathbb{Z}$
$\dot{r}$	::=	$\dot{e} \mid \mathtt{null} \mid \mathtt{bounds}$
$\dot{E}$	::=	$\square \mid \mathtt{let}\ x = \dot{E}\ \mathtt{in}\ \dot{e} \mid \mathtt{ret}(x, i, \dot{E})$
	$\mid$	$\mathtt{if}\ (\dot{E})\ \dot{e}\ \mathtt{else}\ \dot{e} \mid \mathtt{strlen}(\dot{E})$
	$\mid$	$\mathtt{malloc}(\dot{E}, \mid) f(\overline{\dot{E}}) \mid \dot{E} \circ \dot{a} \mid n \circ \dot{E}$
	$\mid$	$*\dot{E} \mid *\dot{E} = \dot{a} \mid *n = \dot{E} \mid x = \dot{E}$
$\overline{\dot{E}}$	::=	$\dot{E} \mid n, \overline{\dot{E}} \mid \overline{\dot{E}}, \dot{a}$

Figure 24: CoreC Semantic Defs

CoreC does not include any annotations. We remove structs from CoreC because we can always statically convert expressions of the form $\&n : \tau \rightarrow f$ into $n + n_f$, where n_f is the statically determined offset of f within the struct. We ellide the semantics of CoreC because it is self-evident and mirrors the semantics CoreChkCBox. The difference is that in CoreC, only `bounds` and `null` can step into an error state. All failed dereferences and assignments would result in a stuck state and therefore we rely on the compiler to explicitly insert checks for checked pointers.

Fig. 27 and Fig. 28 shows the rules for the compilation judgment for expressions,

$$\Gamma; \rho \vdash e \gg \dot{C}, \dot{a}$$

The judgment is presented differently from the one in Sec. 4, which was simplified for presentation purposes. First, we remove Θ and m because these parameters are only used for checking and have no impact on compilation. Second, the judgment includes two outputs, a closure $\dot{C}$ and an atom expression $\dot{a}$, instead of a single CoreC expression $\dot{e}$. $\dot{C}$ can be intuitively understood as a partially constructed program or context. Whereas $\dot{E}$ is used for evaluation, $\dot{C}$ is used purely as a device for compilation. As an example, when compiling $(1 : \mathtt{int}) + (2 : \mathtt{int})$, we would first create a fresh variable x, and then produce two outputs:

$$\dot{C} = \mathtt{let}\ x = 1 + 2\ \mathtt{in}\ \square$$

$$\dot{a} = x$$

To obtain the compiled expression $\dot{e}$, we plug $\dot{a}$ into $\dot{C}$ using the usual notation $\dot{C}[\dot{a}]$. We can also use $\dot{C}$ to represent runtime checks, which usually take the form $\mathtt{let}\ x = \dot{c}\ \mathtt{in}\ \square$, where $\dot{c}$ contains the check whose evaluation must not trigger `bounds` or `null` for the program to continue (see Fig. 26 for the metafunctions that create those checks).

This unconventional output format enables us to separate the evaluation of the term and the computation that relies on the term's evaluated result. Since effects and reduction (except for variables) happen only within closures, we can precisely control the order in which effects and evaluation happen by composing the contexts in a specific order. Given two closures $\dot{C}_1$ and $\dot{C}_2$, we write $\dot{C}_1[\dot{C}_2]$ to denote the meta operation of plugging $\dot{C}_2$ into $\dot{C}_1$. We also use $\dot{C}_{a;b;c}$ as a shorthand for $\dot{C}_a[\dot{C}_b[\dot{C}_c]]$. In the C-Ind rule, we first evaluate the expressions that correspond to e_1 and e_2 through $\dot{C}_1$ and $\dot{C}_2$, and then perform a null check and an addition through $\dot{C}_n$ and $\dot{C}_3$. Finally, we dereference the result through $\dot{C}_4$ before returning the pair $\dot{C}_4, \dot{x}_4$, propagating the flexibility to the compilation rule that recursively calls C-Ind.

Fig. 26 shows the metafunctions that create closures representing dynamic checks. These functions first examine whether the pointer is a checked. If the pointer is unchecked, an empty closure $\square$ will be returned, because there is no need to perform a check. For bounds checking, there is a special case for NT-array pointers, where the bounds are retrived from the shadow variables (found by looking up ρ) on the stack rather than using the bounds specified in the type annotation. This is how we achieve the same precise runtime behavior as CoreChkCBox in our compiled expressions.

Fig. 25 shows the metafunctions related to bounds widening. $\vdash_{extend}$ takes ρ, a checked NT-array pointer variable x, and its bounds (b_l, b_h) as inputs, and returns an extended ρ' that maps x to two fresh variables x_l, x_h, together with a closure $\dot{C}$ that initializes x_l and x_h to b_l and b_h respectively. This function is used in the C-LET rule to extend ρ before compiling the body of the `let` binding. The updated ρ' can be used for generating precise bounds checks, and for inserting expressions that can potentially widen the upper bounds, as seen in the $\vdash_{widenstr}$ metafunction used in the C-STR compilation rule.

$$\frac{x_l, x_h = \texttt{fresh} \qquad \rho' = \rho[x \mapsto (x_l, x_h)] \qquad \dot{C} = \texttt{let } x_l = b_l \texttt{ in let } x_h = b_h \texttt{ in } \square}{\dot{C}, \rho' = \ \vdash_{extend} \rho, x, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt}}$$

$$\frac{x_l, x_h = \rho(x) \qquad x_w = \texttt{fresh} \qquad \dot{C} = \texttt{let } x_w = \texttt{if } (x_h)\ 0 \texttt{ else } x_h = 1 \texttt{ in } \square}{\dot{C} = \ \vdash_{widenderef} \rho, x, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt}} \qquad \frac{e \notin dom(\rho)}{\square = \ \vdash_{widenstr} \rho, e, \dot{a}, \texttt{ptr}^m\ [\beta\ \tau]_{nt}}$$

$$\frac{x_l, x_h = \rho(e) \qquad x_a = \texttt{fresh} \qquad \dot{C} = \texttt{let } x_a = \texttt{if } (\dot{a} \leq x_h)\ 0 \texttt{ else } x_h = \dot{a} \texttt{ in } \square}{\dot{C} = \ \vdash_{widenstr} \rho, e, \dot{a}, \texttt{ptr}^c\ [\beta\ \tau]_{nt}}$$

Figure 25: Metafunctions for Widening

$$\frac{x = \texttt{fresh} \qquad \dot{C} = \texttt{let } x = \texttt{if } (\dot{a})\ 0 \texttt{ else null in } \square}{\dot{C} = \ \vdash_{null} \dot{a}, c} \qquad \square = \ \vdash_{null} \dot{a}, u$$

$$\square = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^u\ [\beta\ \tau]_\kappa, \dot{a}$$

$$\frac{\begin{array}{c} x_l, x_h = \rho(e) \\ x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_{cl} = \texttt{let } x_{cl} = \texttt{if } (x_l \leq \dot{a})\ 0 \texttt{ else bounds in } \square \qquad \dot{C}_{ch} = \texttt{let } x_{ch} = \texttt{if } (\dot{a} \leq x_h)\ 0 \texttt{ else bounds in } \square \end{array}}{\dot{C}_{cl;ch} = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^c\ [\beta\ \tau]_\kappa, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_h, x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_l = \texttt{let } x_l = b_l \texttt{ in } \square \qquad \dot{C}_h = \texttt{let } x_h = b_h \texttt{ in } \square \\ \dot{C}_{cl} = \texttt{let } x_{cl} = \texttt{if } (x_l \leq \dot{a})\ 0 \texttt{ else bounds in } \square \qquad \dot{C}_{ch} = \texttt{let } x_{ch} = \texttt{if } (\dot{a} \leq x_h)\ 0 \texttt{ else bounds in } \square \end{array}}{\dot{C}_{l;h;cl;ch} = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt}, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_h, x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_l = \texttt{let } x_l = b_l \texttt{ in } \square \qquad \dot{C}_h = \texttt{let } x_h = b_h \texttt{ in } \square \\ \dot{C}_{cl} = \texttt{let } x_{cl} = \texttt{if } (x_l \leq \dot{a})\ 0 \texttt{ else bounds in } \square \qquad \dot{C}_{ch} = \texttt{let } x_{ch} = \texttt{if } (x_h \leq \dot{a}) \texttt{ bounds else } 0 \texttt{ in } \square \end{array}}{\dot{C}_{l;h;cl;ch} = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^c\ [(b_l, b_h)\ \tau], \dot{a}}$$

$$\square = \ \vdash_{boundsW} \rho, e, \texttt{ptr}^u\ [\beta\ \tau]_\kappa, \dot{a}$$

$$\frac{\begin{array}{c} x_l, x_h = \rho(e) \\ x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_{cl} = \texttt{let } x_{cl} = \texttt{if } (x_l \leq \dot{a})\ 0 \texttt{ else bounds in } \square \qquad \dot{C}_{ch} = \texttt{let } x_{ch} = \texttt{if } (\dot{a} \leq x_h)\ 0 \texttt{ else bounds in } \square \end{array}}{\dot{C}_{cl;ch} = \ \vdash_{boundsW} \rho, e, \texttt{ptr}^c\ [\beta\ \tau]_\kappa, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_h, x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_l = \texttt{let } x_l = b_l \texttt{ in } \square \qquad \dot{C}_h = \texttt{let } x_h = b_h \texttt{ in } \square \\ \dot{C}_{cl} = \texttt{let } x_{cl} = \texttt{if } (x_l \leq \dot{a})\ 0 \texttt{ else bounds in } \square \qquad \dot{C}_{ch} = \texttt{let } x_{ch} = \texttt{if } (x_h \leq \dot{a}) \texttt{ bounds else } 0 \texttt{ in } \square \end{array}}{\dot{C}_{l;h;cl;ch} = \ \vdash_{boundsW} \rho, e, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_\kappa, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_l', x_h, x_h' = \texttt{fresh} \qquad \dot{C}_1 = \texttt{let } x_l = b_l \texttt{ in let } x_h = b_h \texttt{ in } \square \\ \dot{C}_2 = \texttt{let } x_l' = b_l' \texttt{ in let } x_h' = b_h' \texttt{ in } \square \qquad \dot{C}_3 = \texttt{if } (x_l' \leq x_l)\ \square \texttt{ else bounds} \qquad \dot{C}_4 = \texttt{if } (x_h \leq x_h')\ \square \texttt{ else bounds} \end{array}}{\dot{C}_{1;2;3;4} = \ \vdash_{boundsD} \rho, e, \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa, \texttt{ptr}^m\ [(b_l', b_h')\ \tau]_\kappa}$$

$$\frac{\begin{array}{c} x_l', x_h' = \rho(e) \qquad x_l, x_h = \texttt{fresh} \\ \dot{C}_1 = \texttt{let } x_l = b_l \texttt{ in let } x_h = b_h \texttt{ in } \square \qquad \dot{C}_2 = \texttt{if } (x_l' \leq x_l)\ \square \texttt{ else bounds} \qquad \dot{C}_3 = \texttt{if } (x_h \leq x_h')\ \square \texttt{ else bounds} \end{array}}{\dot{C}_{1;2;3} = \ \vdash_{boundsD} \rho, e, \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa, \texttt{ptr}^m\ [(b_l', b_h')\ \tau]_\kappa}$$

Figure 26: Metafunctions for Dynamic Checks

C-CONST
$$\Gamma; \rho \vdash n : \tau \gg \square, n : \tau$$

C-VAR
$$\frac{x : \tau \in \Gamma}{\Gamma; \rho \vdash x \gg \square, x : \tau}$$

C-CAST
$$\frac{\Gamma; \rho \vdash e \gg \dot{C}, \dot{a} : \tau'}{\Gamma; \rho \vdash (\tau)e \gg \dot{C}, \dot{a} : \tau}$$

C-DYNCAST
$$\frac{\Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a} : \texttt{ptr}^m\ [\beta'\ \tau]_\kappa \qquad \dot{C}_b = \ \vdash_{boundsD}\ \rho, e, \texttt{ptr}^m\ [\beta\ \tau]_\kappa, \texttt{ptr}^m\ [\beta'\ \tau]_\kappa}{\Gamma; \rho \vdash \langle \texttt{ptr}^m\ [\beta\ \tau]_\kappa \rangle e \gg \dot{C}_{1;b}, \dot{a} : \texttt{ptr}^m\ [\beta\ \tau]_\kappa}$$

C-STR
$$\frac{\Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^m\ [\beta\ \tau_a]_{nt} \qquad \dot{C}_n = \ \vdash_{null}\ \dot{a}_1, m \qquad \dot{C}_b = \ \vdash_{boundsR}\ \rho, \dot{a}_1, \texttt{ptr}^m\ [\beta\ \tau_a]_{nt}, 0 \qquad x_2 = \texttt{fresh} \qquad \dot{C}_2 = \texttt{let}\ x_2 = \texttt{strlen}(\dot{a}_1)\ \texttt{in}\ \square \qquad \dot{C}_w = \ \vdash_{widenstr}\ \rho, e, \dot{a}_1, \texttt{ptr}^m\ [\beta\ \tau_a]_{nt}}{\Gamma; \rho \vdash \texttt{strlen}(e) \gg \dot{C}_{1;n;b;2;w}, x_2 : \texttt{int}}$$

C-LETSTR
$$\frac{\Gamma(y) = \texttt{ptr}^c\ [(b_l, b_h)\ \tau_a]_{nt} \qquad x \notin FV(\tau) \qquad \Gamma; \rho \vdash \texttt{strlen}(y) \gg \dot{C}_1, \dot{a}_1 : \texttt{int} \qquad \dot{C}_2 = \texttt{let}\ x = \dot{a}_1\ \texttt{in}\ \square \qquad \Gamma[x \mapsto \texttt{int}, y \mapsto [\texttt{ptr}^c\ [(b_l, x)\ \tau_a]_{nt}]]; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau}{\Gamma; \rho \vdash \texttt{let}\ x = \texttt{strlen}(y)\ \texttt{in}\ e \gg \dot{C}_{1;2;3}, \dot{a}_3 : \tau}$$

C-IF
$$\frac{\Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a}_1 : \tau \qquad \Gamma; \rho \vdash e_1 \gg \dot{C}_2, \dot{a}_2 : \tau_2 \qquad \Gamma; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau_3 \qquad x_4 = \texttt{fresh} \qquad \dot{C}_4 = \texttt{let}\ x_4 = \texttt{if}\ (\dot{a}_1)\ \dot{C}_2[\dot{a}_2]\ \texttt{else}\ \dot{C}_3[\dot{a}_3]\ \texttt{in}\ \square}{\Gamma; \rho \vdash \texttt{if}\ (e_1)\ e_2\ \texttt{else}\ e_3 \gg \dot{C}_{1;4}, x_4 : \tau_2 \sqcup \tau_3}$$

C-IFNT
$$\frac{\Gamma; \rho \vdash x : \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt} \qquad b_h = 0 \Rightarrow \Gamma' = \Gamma[x \mapsto \texttt{ptr}^c\ [(b_l, 1)\ \tau]_{nt}] \qquad b_h \neq 0 \Rightarrow \Gamma' = \Gamma \qquad \Gamma; \rho \vdash *x \gg \dot{C}_1, \dot{a}_1 : \tau_1 \qquad \Gamma'; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \tau_2 \qquad \Gamma; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau_3 \qquad \dot{C}_w = \ \vdash_{widenderef}\ \rho, x, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt} \qquad x_4 = \texttt{fresh} \qquad \dot{C}_4 = \texttt{let}\ x_4 = \texttt{if}\ (\dot{a}_1)\ \dot{C}_{2;w}[\dot{a}_2]\ \texttt{else}\ \dot{C}_3[\dot{a}_3]\ \texttt{in}\ \square}{\Gamma; \rho \vdash \texttt{if}\ (*x)\ e_1\ \texttt{else}\ e_2 \gg \dot{C}_{1;4}, x_4 : \tau_1 \sqcup \tau_2}$$

C-LET
$$\frac{(x \in FV(\tau') \Rightarrow e_1 \in Bound) \qquad \Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \tau_1 \qquad \dot{C}_2, \rho' = \ \vdash_{extend}\ \rho, x, \tau_1 \qquad \dot{C}_3 = \texttt{let}\ x = \dot{a}_1\ \texttt{in}\ \square \qquad \Gamma[x \mapsto \tau]; \rho' \vdash e_4 \gg \dot{C}_4, \dot{a}_4 : \tau_4}{\Gamma; \rho' \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_4 \gg \dot{C}_{1;2;3;4}, \dot{a}_4 : \tau_4[\tau_1 = \texttt{int} \Rightarrow x \mapsto e_1]}$$

C-RET
$$\frac{\Gamma(x) \neq \bot \qquad \Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a}_1 : \tau \qquad x_2 = \texttt{fresh} \qquad \mu \gg \dot{\mu} \qquad \dot{C}_2 = \texttt{let}\ x_2 = \texttt{ret}(x, \dot{\mu}, \dot{C}_1[\dot{a}_1])\ \texttt{in}\ \square}{\Gamma; \rho \vdash \texttt{ret}(x, \mu, e) \gg \dot{C}_2, x_2 : \tau}$$

C-FUN
$$\frac{\Xi(f) = \tau\ (\overline{x} : \overline{\tau})\ e \qquad (\forall e_i \in \overline{e}\ \ \tau_i \in \overline{\tau}\ .\ \Gamma; \rho \vdash e_i \gg \dot{C}_i, \dot{a}_i : \qquad \tau_i' \wedge \tau_i' \sqsubseteq \tau_i[\overline{e}/\overline{x}]) \qquad x_f = \texttt{fresh} \qquad \dot{C}_f = \texttt{let}\ x_f = f(\overline{a})\ \texttt{in}\ \square}{\Gamma; \rho \vdash f(\overline{e}) \gg \overline{\dot{C}}[\dot{C}_f], x_f : \tau[\overline{e}/\overline{x}]}$$

C-DEF
$$\frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^m\ \tau \qquad \dot{C}_n = \ \vdash_{null}\ \dot{a}_1, m \qquad x_2 = \texttt{fresh} \qquad \dot{C}_2 = \texttt{let}\ x_2 = *\dot{a}_1\ \texttt{in}\ \square}{\Gamma; \rho \vdash *e_1 \gg \dot{C}_{1;n;2}, x_2 : \tau}$$

C-DEFARR
$$\frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa \qquad \dot{C}_n = \ \vdash_{null}\ \dot{a}_1, m \qquad \dot{C}_b = \ \vdash_{boundsR}\ \rho, e_1, \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa, 0 \qquad x_2 = \texttt{fresh} \qquad \dot{C}_2 = \texttt{let}\ x_2 = *\dot{a}_1\ \texttt{in}\ \square}{\Gamma; \rho \vdash *e_1 \gg \dot{C}_{1;n;b;2}, x_2 : \tau}$$

C-MAC
$$\frac{\dot{C}_1, \dot{a}_1 = \texttt{sizeof}(\omega) \qquad x_2 = \texttt{fresh} \qquad \dot{C}_2 = \texttt{let}\ x_2 = \texttt{malloc}(\dot{a}_1, \ )\texttt{in}\ \square}{\Gamma; \rho \vdash \texttt{malloc}(\omega, \gg)\dot{C}_{1;2}, x_2 : \texttt{ptr}^c\ \omega}$$

Figure 27: Compilation

C-ADD

$$\frac{\Gamma;\rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{int} \qquad \Gamma;\rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \texttt{int} \qquad x_3 = \texttt{fresh} \qquad \dot{C}_3 = \texttt{let } x_3 \texttt{ = } \dot{a}_1 \texttt{ + } \dot{a}_2 \texttt{ in } \square}{\Gamma;\rho \vdash \dot{C}_3, x_3 : \texttt{int}}$$

C-IND

$$\frac{\begin{array}{c}\Gamma;\rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^m\ [\beta\ \tau]_\kappa \qquad \Gamma;\rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \texttt{int} \qquad \dot{C}_n = \ \vdash_{null} \dot{a}_1, m \\ \dot{C}_b = \ \vdash_{boundsR} \rho, e_1, \texttt{ptr}^m\ [\beta\ \tau]_\kappa, \dot{a}_2 \qquad x_3, x_4 = \texttt{fresh} \qquad \dot{C}_3 = \texttt{let } x_3 \texttt{ = } \dot{a}_1 \texttt{ + } \dot{a}_2 \texttt{ in } \square \qquad \dot{C}_4 = \texttt{let } x_4 \texttt{ = } \texttt{*}x_3 \texttt{ in } \square\end{array}}{\Gamma;\rho \vdash \texttt{*}(e_1 \texttt{ + } e_2) \gg \dot{C}_{1;2;n;3;b;4}, x_4 : \tau}$$

C-ASSIGN

$$\frac{\begin{array}{c}\Gamma;\rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^{m'}\ \tau \\ \dot{C}_n = \ \vdash_{null} \dot{a}_1, m \qquad \Gamma;\rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \tau' \qquad \tau' \sqsubseteq \tau \qquad x_3 = \texttt{fresh} \qquad \dot{C}_3 = \texttt{let } x_3 \texttt{ = } \texttt{*}\dot{a}_1 \texttt{ = } \dot{a}_2 \texttt{ in } \square\end{array}}{\Gamma;\rho \vdash \texttt{*}e_1 \texttt{ = } e_2 \gg \dot{C}_{1;2;n;3}, x_3 : \tau}$$

C-ASSIGNARR

$$\frac{\begin{array}{c}\Gamma;\rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad \dot{C}_n = \ \vdash_{null} \dot{a}_1, m \qquad \dot{C}_b = \ \vdash_{boundsW} \rho, e_1, \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa, 0 \\ \Gamma;\rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \tau' \qquad x_3 = \texttt{fresh} \qquad \dot{C}_3 = \texttt{let } x_3 \texttt{ = } \texttt{*}\dot{a}_1 \texttt{ = } \dot{a}_2 \texttt{ in } \square \qquad \tau' \sqsubseteq \tau\end{array}}{\Gamma;\rho \vdash \texttt{*}e_1 \texttt{ = } e_2 \gg \dot{C}_{1;2;n;b;3}, x_3 : \tau}$$

C-INDASSIGN

$$\frac{\begin{array}{c}\Gamma;\rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^m\ [\beta\ \tau]_\kappa \\ \Gamma;\rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \texttt{int} \qquad \dot{C}_n = \ \vdash_{null} \dot{a}_1, m \qquad \dot{C}_b = \ \vdash_{boundsW} \rho, e_1, \texttt{ptr}^m\ [\beta\ \tau]_\kappa, \dot{a}_2 \\ \Gamma;\rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau' \qquad x_4, x_5 = \texttt{fresh} \qquad \dot{C}_4 = \texttt{let } x_4 \texttt{ = } \dot{a}_1 \texttt{ + } \dot{a}_2 \texttt{ in } \square \qquad \dot{C}_5 = \texttt{let } x_5 \texttt{ = } \texttt{*}x_4 \texttt{ = } x_3 \square \texttt{ in } \tau' \sqsubseteq \tau\end{array}}{\Gamma;\rho \vdash \texttt{*}(e_1 \texttt{ + } e_2) \texttt{ = } e_3 \gg \dot{C}_{1;2;n;3;4;b;5} : \tau}$$

C-STRUCT

$$\frac{\begin{array}{c}\Gamma;\rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \texttt{ptr}^m\ \texttt{struct}\ T \\ D(T) = \tau_0\ f_0 \ldots; \tau_j\ f; \ldots \qquad \dot{C}_n = \ \vdash_{null} \dot{a}_1, m \qquad x_2 = \texttt{fresh} \qquad \dot{C}_2 = \texttt{let } x_2 \texttt{ = } \dot{a}_1 \texttt{ + } \dot{j} \texttt{ in } \square\end{array}}{\Gamma;\rho \vdash \&e_1 \rightarrow f \gg \dot{C}_2, x_2 : \texttt{ptr}^m\ \tau_f}$$

C-UNCHECKED

$$\frac{\Gamma;\rho \vdash e \gg \dot{C}, \dot{a} : \tau}{\Gamma;\rho \vdash \texttt{unchecked}(e)\{\gg\}\dot{C}, \dot{a} : \tau}$$

Figure 28: Compilation (Continued)