A Formal Model of Checked C

Liyi Li, Yiyun Liu†, Deena Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks
University of Maryland †University of Pennsylvania

***Abstract*—We present a formal model of Checked C, a dialect of C that aims to enforce spatial memory safety. Our model pays particular attention to the semantics of dynamically sized, potentially null-terminated arrays. We formalize this model in Coq, and prove that any spatial memory safety errors can be *blamed* on portions of the program labeled *unchecked*; this is a Checked C feature that supports incremental porting and backward compatibility. While our model's operational semantics uses annotated ("fat") pointers to enforce spatial safety, we show that such annotations can be safely erased. Using PLT Redex we formalize an executable version of our model and a compilation procedure to an untyped C-like language, as well as use randomized testing to validate that generated code faithfully simulates the original. Finally, we develop a custom random generator for well-typed and almost-well-typed terms in our Redex model, and use it to search for inconsistencies between our model and the Clang Checked C implementation. We find these steps to be a useful way to co-develop a language (Checked C is still in development) and a core model of it.**

I. Introduction

The C programming language remains extremely popular despite the emergence of new, modern languages. Unfortunately, C programs lack spatial memory safety, which makes them susceptible to a host of devastating vulnerabilities, including buffer overflows and out-of-bounds reads/writes. Despite their long history, buffer overflows and other spatial safety violations are among the most prevalent and dangerous vulnerabilities on the Internet today [27].

Several industrial and research efforts—including CCured [20], Softbound [19], and ASAN [24]—have explored means to compile C programs to automatically enforce spatial safety. These approaches all impose performance overheads deemed too high for deployment use. Recently, Elliott et al. [4] introduced Checked C, an open-source extension to C with new types and annotations whose use can ensure a program's spatial safety. Importantly, Checked C supports development that is incremental and compositional. Code regions (e.g., functions or whole files) designated as *checked* enforce spatial safety in a manner preserved by composition with other checked regions. But not all regions must be checked: Checked C's annotated *checked pointers* are binary-compatible with legacy pointers, and may coexist in the same code, which permits a deliberate (and semi-automated) refactoring process. Parts of the FreeBSD kernel have been successfully ported to Checked C [3], and overall, performance overhead seems low enough for practical deployment.

While Checked C promises to enforce spatial safety, we might wonder whether its design and implementation deliver on this promise, or even what "spatial safety" means when a program contains both checked and unchecked code. In prior work, Ruef et al. [23] developed a core formalization of Checked C and with it proved a *soundness* theorem for checked code: any stuck (i.e., ill-defined) state reached by a well-typed program amounts to a spatial safety violation; such a state can always be attributed to, i.e., *blamed on*, the execution of code that is not in a checked region. While their work is a good start, it fails to model important aspects of Checked C's functionality, particularly those involving pointers to arrays. In this paper, we cover this gap, making three main contributions.

Dynamically bounded and null-terminated arrays. Our first contribution is a core formalism called CoreChkC, which extends Ruef et al. [23] with several new features, most notably *dynamically bounded arrays* (Section III). Dynamically bounded arrays are those whose size is known only at run time, as designated by in-scope variables using dependent types. A pointer's accessible memory is bounded both above and below, to admit arbitrary pointer arithmetic.

CoreChkC also models *null-terminated* arrays, a kind of dynamically bounded array whose upper bound defines the array's *minimum* length—additional space is available up to a null terminator. For example, the Checked C type `nt_array_ptr<char> p:count(n)` says that `p` has length *at least* `n` (excluding the null terminator), but further capacity is present if `p[n]` is not null. Checked C (and CoreChkC) supports flow-sensitive *bounds widening*: statements of the form `if (*p) s`, where `p`'s type is `nt_array_ptr<T> count(0)`, typecheck statement s under the assumption that `p` has type `nt_array_ptr<T> count(1)`, i.e., one more than it was, since the character at the current-known length is non-null. Similarly, the call `n = strlen(p)` will widen `p`'s bounds to `n`. Subtyping permits treating null-terminated arrays as normal arrays of the same size (which does not include, and thereby protects, the null terminator).[1]

We prove, in Coq, a blame theorem for CoreChkC. As far as we are aware, ours is the first formalized type system and proof of soundness for pointers to null-terminated arrays with expandable bounds.

Sound compilation of checked pointers. Our second contribution is a formalization of bounds-check insertion for array accesses (Section IV). Our operational semantics annotates each pointer with metadata that describes its bounds, and the assignment and dereference rules have premises to confirm the access is in bounds. An obvious compilation scheme (taken by Cyclone [7, 10], CCured [20], and earlier works) would be to translate annotated pointers to multi-word objects: one word for the pointer, and 1-2 words to describe its lower

[1] See Sec. VI for a careful comparison of Ruef et al. [23] and CoreChkC.

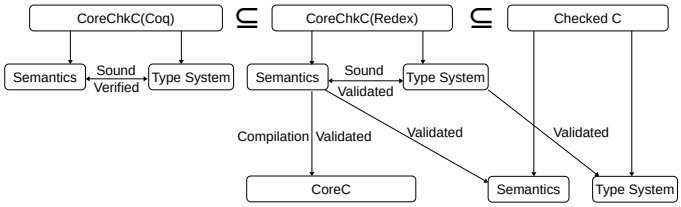

Fig. 1: CORECHKC models' relationship to Checked C

and upper bounds. Inserted checks reference these bounds. While convenient, such "fat" pointers are expensive, and break backward binary compatibility with legacy pointers.

To show that pointer annotations can be safely erased, and thus fat pointers are not needed, we formalize a translation of CORECHKC to COREC, which is an untyped version of CORECHKC that drops metadata annotations, and lacks bounds/null checks in the semantics rules. Instead, the compilation process inserts null/bounds checks explicitly, leveraging compile-time type information. While we do not definitively prove it, we provide strong evidence that compilation is correct. We use PLT Redex [6] to mechanize (a generalization of) CORECHKC, COREC, and compilation between the two, and we use randomized testing to validate that the compiled program *simulates* the original. In addition to demonstrating the technical point that metadata annotations in the CORECHKC formalism do not necessitate fat pointers, compilation also sheds light on the actual Checked C compilation process.

As far as we are aware, CORECHKC is the first formalism to cleanly separate bounds-checking compilation from the core semantics; prior work [2, 29] merged the two, conflating *meaning* with *mechanism*. In carrying out the formalization, we discovered that our compilation approach for null-terminated array pointers is more expressive than that proposed in the Checked C specification [25] (Section IV-B); we would not have discovered this improvement had we not separated checks from semantics.

Model-based randomized testing. Our third and final contribution is a strategy and implementation of model-based randomized testing (Section V). To check the correctness of our formal model, we compare the behavior between the existing Clang Checked C implementation and our own model. This is done by a conversion tool that converts expressions from CORECHKC into actual Checked C code that can be compiled by the Clang Checked C compiler. We build a random generator of programs largely based on the typing rules of CORECHKC and make sure that, both statically and dynamically, CORECHKC and Clang Checked C are consistent after conversion. This helped rapidly prototype the model and uncovered several issues in the Checked C compiler.

Summary Visualization. The relationship among our contributions is visualized in Fig. 1. With the Coq model of CORECHKC we prove soundness (and with it, *blame*) of the Checked C type system and semantics. With the Redex model, we use randomized testing to validate both type soundness and compilation correctness, where the latter shows how compilation need not output fat pointers despite the use of pointer annotations in the CORECHKC model. The Redex CORECHKC model is also the basis of randomized testing of the correctness of the Checked C compiler implementation, both its type checker and the semantics of its emitted code, at least for the subset of the language in the Redex model. The Redex model's syntax is slightly richer than the Coq version: conditional guards and function arguments may be arbitrary expressions, where the Coq version limits them to constants and variables, making handling of dependent types a bit simpler. We find a useful synergy between the Coq and Redex models for carrying out a language development. The richer, executable Redex model is useful for quickly modeling and testing new features, both formally and against a real implementation. Once solidified, new features can be added to the Coq model (perhaps somewhat simplified) for final proofs of correctness.

We begin with a review of Checked C (Section II), present our main contributions (Sections III–V), and conclude with a discussion of related and future work (Sections VI, VII). All code and proof artifacts (both for Coq and Redex) can be found at https://github.com/plum-umd/checkedc.

II. CHECKED C OVERVIEW

This section describes Checked C, which extends C with new pointer types and annotations that ensure spatial safety. Development of Checked C was initiated by Microsoft Research in 2015 but starting in late 2021 was forked and is now actively managed by the Secure Software Development Project (SSDP). Details can be found in a prior overview [4] or the full specification [25]. Checked C is implemented as an extension of Clang/LLVM and the SSDP fork is freely available at https://github.com/secure-sw-dev.

A. Checked Pointer Types

Checked C introduces three varieties of *checked pointer*:

- `ptr<`T`>` types a pointer that is either null or points to a single object of type T.
- `array_ptr<`T`>` types a pointer that is either null or points to an array of T objects. The array width is defined by a *bounds* expression, discussed below.
- `nt_array_ptr<`T`>` is like `array_ptr<`T`>` except that the bounds expression defines the *minimum* array width—additional objects may be available past the upper bound, up to a null terminator.

A bounds expression used with the latter two pointer types has three forms:

- `count(`e`)` where e defines the array's length. Thus, if pointer p has bounds `count(n)` then the accessible memory is in the range $[p, p+\texttt{n}]$. Bounds expression e must be side-effect free and may only refer to variables whose addresses are not taken, or adjacent `struct` fields.
- `byte_count(`e`)` is like `count`, but expresses arithmetic using bytes, no objects; i.e., `count(`e`)` used

```
1  nt_array_ptr<const char>
2  parse_utf16_hex(nt_array_ptr<const char> s,
3                  ptr<uint> result) {
4   int x1, x2, x3, x4;
5   if (s[0] != 0) { x1 = hex_char_to_int(s[0]);
6   if (s[1] != 0) { x2 = hex_char_to_int(s[1]);
7   if (s[2] != 0) { x3 = hex_char_to_int(s[2]);
8   if (s[3] != 0) { x4 = hex_char_to_int(s[3]);
9   if (x1 != -1 && x2 != -1 && x3 != -1 && x4 != -1){
10    *result = (uint)((x1<<12)|(x2<<8)|(x3<<4)|x4);
11    return s+4;
12   ...// several } braces
13  }
14  return 0;
15 }
16 void parse(nt_array_ptr<const char> s,
17            array_ptr<uint> p : count(n),
18            int n) {
19  array_ptr<uint> q : bounds(p,p+n) = p;
20  while (s && q < p+n) {
21    array_ptr<uint> r : count(1) =
22      dyn_bounds_cast<array_ptr<uint>>(q,count(1));
23    s = parse_utf16_hex(s,r);
24    q++;
25  }
26 }
```

Fig. 2: Parsing a String of UTF16 Hex Characters in Checked C

for `array_ptr<`T`>` is equivalent to `byte_count(`$e \times$ `sizeof(`T`))`

- `bounds(`e_l`,`e_h`)` where e_l and e_h are pointers that bound the accessible region $[e_l, e_h)$ (the expressions are similarly restricted). Bounds `count(`e`)` is shorthand for `bounds(`$p, p + e$`)`. This most general form of bounds expression is useful for supporting pointer arithmetic.

Dropping the bounds expression on an `nt_array_ptr` is equivalent to the bounds being `count(0)`.

The Checked C compiler will instrument loads and stores of checked pointers to confirm the pointer is non-null, and the access is within the specified bounds. For pointers p of type `nt_array_ptr<`T`>`, such a check could spuriously fail if the index is past p's specified upper bound, but before the null terminator. To address this problem, Checked C supports *bounds widening*. If p's bounds expression is `bounds(`e_l`,`e_h`)` a program may read from (but not write to) e_h; when the compiler notices that a non-null character is read at the upper bound, it will extend that bound to $e_h + 1$.

B. Example

Fig. 2 gives an example Checked C program.[2] The function `parse_utf16_hex` on lines 1-15 takes a null-terminated pointer `s` as its argument, from which it attempts to read four characters. These are interpreted as hex digits and converted to an `uint` returned via parameter `result`. At first, `s` has no specific bounds annotation, which we can interpret as `count(0)`; this means that `s[0]` may be read on line 5. The true branch of the conditional (which extends all the way to the brace on line 13) is thus type-checked with `s` given a *widened* bound of `count(1)`. Likewise, the conditionals on lines 6-8 each widen it one further; the widened pointer (`s+4`) is returned on success.

The `parse` function on lines 16-26 repeatedly invokes `parse_utf16_hex` with its parameter `s`, and fills out array `p` whose declared length is the parameter `n`. Writes happen via pointer `q`, which is updated using pointer arithmetic. We specify its bounds as `bounds(p,p+n)` to support this: even as `q` changes, variables `p` and `n` (and therefore also `q`'s bounds) do not. Converting from an `array_ptr<uint>` to a `ptr<uint>`, done for the call on line 23, requires proving the array has size at least 1. While this is true because of the loop condition `q < p+n`, which is `q`'s upper bound, the compiler is not smart enough to figure this out. To convince it, we manually insert a *dynamic cast* via `dyn_bounds_cast`, which is trusted at compile-time but confirmed with a dynamic check at run-time.

While bounds checks are *conceptually* inserted on every array load and store, many of these are eliminated by LLVM. For example, all of the pointer accesses to `s` on lines 5-8 are proved safe at compile-time, so no bounds checks are inserted for them. Elliott et al. [4] reported average run-time overheads of 8.6% on a pointer-intensive benchmark suite (49.3% in one case); Duan et al. [3] measured no overhead at all on a port of FreeBSD's UDP and IP stacks to Checked C.

C. Other features

Checked C has other features not modeled in this paper. Two in regular use are *interop types*, which ascribe checked pointer types to unported legacy code, notably in libraries; and *generic types* on both functions and `struct`s, for type-safe polymorphism. More details about these can be found in the language specification.

D. Spatial Safety and Backward Compatibility

Checked C is backward compatible with legacy C in the sense that all legacy code will type-check and compile. However, only code that appears in *checked regions*, which we call *checked code*, is spatially safe. Checked regions can be designated at the level of files, functions, or individual code blocks, the first with a `#pragma` and the latter two using the `checked` keyword.[3] Within checked regions, both legacy pointers and certain unsafe idioms (e.g., *variadic* function calls) are disallowed. The code in Fig. 2 satisfies these conditions, and will type-check in a checked region.

How should we think about code that contains both checked and legacy components? Ruef et al. [23] proved, for a simple formalization of Checked C, that *checked code cannot be blamed*: Any spatial safety violation is caused by the execution of unchecked code. In this paper we extend that result to a richer formalization of Checked C.

[2]Ported from the Parson JSON parser, https://github.com/kgabis/parson

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

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

Fig. 3: CoreChkC Syntax

III. Formalization

This section describes our formal model of Checked C, called CoreChkC, making precise its syntax, semantics, and type system. It also develops CoreChkC's meta-theory, including type soundness and the blame theorem.

A. Syntax

The syntax of CoreChkC is given by the expression-based language presented in Fig. 3.

There are two notions of type in CoreChkC. Types τ classify word-sized values including integers and pointers, while types ω classify multi-word values such as arrays, null-terminated arrays, and single-word-size values. Pointer types (`ptr`^{m} ω) 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). Here is the corresponding Checked C syntax for these types:

`array_ptr<`τ`>` `:` `count(`n`)` $\Leftrightarrow$ `ptr`^{c} $[(0, n)\ \tau]$

`nt_array_ptr<`τ`>` `:` `count(`n`)` $\Leftrightarrow$ `ptr`^{c} $[(0, n)\ \tau]_{nt}$

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

CoreChkC expressions include literals ($n : \tau$), variables (x), memory allocation (`malloc`(ω)), let binding (`let` x`=`e_1 `in` e_2), static casts ($(\tau)e$), dynamic casts ($\langle\tau\rangle e$) (assumed at compile-time and verified at run-time, see Sec. II-B), function calls ($f(\overline{e})$), addition (e_1 `+` e_2), pointer dereference and assignment (`*`e) and (`*`e_1 `=` e_2), resp.), unchecked blocks (`unchecked` e), the `strlen` operation (`strlen`(x)), and conditionals (`if` (e) e_1 `else` e_2).

Integer literals n are annotated with a type τ which can be either `int`, or `ptr`^{m} ω in the case n is being used as a heap address (this is useful for the semantics); $0 : $ `ptr`^{m} ω (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 $\langle$`ptr`^{c} $[n\ \tau]\rangle e$.

CoreChkC 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. `struct`s are not in Fig. 3 for space reasons, but they are actually in our model, and developed in the supplemental report [15]. C-style `union`s have no safe typing in Checked C, so we omit them. By default, functions are assumed to be within checked regions; placing the body in an `unchecked` expression relaxes this, and within that, checked regions can be nested via function calls. Bounds are restricted slightly: rather than allowing arbitrary sub-expressions, bounds must be either integer literals or variables plus an integer offset, which accounts for most uses of `bounds` in Checked C programs. CoreChkC bounds are defined as relative offsets, not absolute ones, as in the second part of Fig. 2. We see no technical problem to modeling absolute bounds, but it would be a pervasive change so we have not done so.

We have mechanized two models of CoreChkC, one in Coq and one in PLT Redex [6], which is a semantic engineering framework implemented in Racket. Redex provides direct support for specifying the operational semantics and typing with logical rules, but then automatically makes them executable and subject to randomized testing, which is very useful during development. The model we present in the paper faithfully represents both mechanizations, but there are some differences for presentation purposes. For example, the paper and the Coq model use an explicit stack, whereas the Redex model uses `let` bindings to simulate one (simplifying term generation for randomized testing). The supplemental report [15] outlines the differences between the two models and the paper formalism.

B. Semantics

The operational semantics for CoreChkC 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* mapping addresses (integer literals) to values $n{:}\tau$; for both we ensure $FV(\tau) = \emptyset$. 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

[3] You can also designate *unchecked* regions within checked ones.

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

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

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

Fig. 4: CoreChkC Semantics: Evaluation

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 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. 4. 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. 5, discussed shortly. The second rule handles conditionals $\texttt{if}\ (*x)\ e_2\ \texttt{else}\ e_3$ 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.[4] The *mode* function determines the mode when evaluating e' based on the context E: if the $\square$ occurs within $(\texttt{unchecked}\ E')$ inside E, then the mode is `u`; otherwise, it is `c`. Evaluation contexts E define a standard left-to-right evaluation order. (We explain the $\texttt{ret}(x, \mu, e)$ syntax shortly.)

Fig. 5 shows selected rules for the computation relation; we explain them with the help of the example in Fig. 6, which defines a safe version of `strcat` (using actual Checked C syntax). The function takes a target pointer `dst` of capacity `n`, where the first null character (determined by `strlen`) is at index `x` where $0 \leq$`x`$\leq n$. It concatenates the `src` buffer to the end of `dst` as long as `dst` has sufficient space.

Pointer accesses. The rules for dereference and assignment operations—S-Def, S-DefNull, S-DefNTArray, and S-AssignArr—illustrate how the semantics checks bounds. Rule S-DefNull transitions attempted null-pointer dereferences to `null`, whereas S-Def dereferences a 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. 4); it does likewise when `bounds` is returned (see below).

S-AssignArr 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. 6. 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 evaluates to a configuration whose stack is φ extended with a binding for x, and whose expression is $\texttt{ret}(x, \varphi(x), e))$ 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. 6. 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. 6 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 eval-

[4]This approach is that of the PLT Redex model of CoreChkC; the Coq development uses a slightly simpler syntax to achieve the same effect.

S-DEF

$$\frac{\mathcal{H}(n) = n_a : \tau_a}{(\varphi, \mathcal{H}, *n : \texttt{ptr}^m\ \tau) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}$$

S-DEFNULL

$$(\varphi, \mathcal{H}, *0 : \texttt{ptr}^c\ \omega) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})$$

S-DEFNTARRAY

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

S-ASSIGNARR

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

S-CAST

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

S-DYNCAST

$$\frac{\varphi(\texttt{ptr}^m\ [\beta\ \tau]_\kappa) = \texttt{ptr}^m\ [(n_l', n_h')\ \tau_b]_\kappa \qquad n_l' \le n_l \qquad n_h \le n_h'}{(\varphi, \mathcal{H}, \langle \texttt{ptr}^m\ [\beta\ \tau]_\kappa \rangle n : \texttt{ptr}^m\ [(n_l, n_h)\ \tau_a]_\kappa) \longrightarrow (\varphi', \mathcal{H}', n : \texttt{ptr}^m\ [(n_l', n_h')\ \tau_b]_\kappa)}$$

S-DYNCASTBOUND

$$\frac{\varphi(\texttt{ptr}^c\ [\beta\ \tau]_\kappa) = \texttt{ptr}^c\ [(n_l', n_h')\ \tau_b]_\kappa \qquad n_l' > n_l \vee n_h > n_h'}{(\varphi, \mathcal{H}, \langle \texttt{ptr}^c\ [\beta\ \tau]_\kappa \rangle n : \texttt{ptr}^c\ [(n_l, n_h)\ \tau_a]_\kappa) \longrightarrow (\varphi', \mathcal{H}', \texttt{bounds})}$$

S-LET

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

S-RET

$$(\varphi, \mathcal{H}, \texttt{ret}(x, \mu, n : \tau)) \longrightarrow (\varphi[x \mapsto \mu], \mathcal{H}, n : \tau)$$

S-FUN

$$\frac{\Xi(f) = \tau\ (\overline{x} : \overline{\tau})\ e}{(\varphi, \mathcal{H}, f(\overline{n} : \overline{\tau_a})) \longrightarrow (\varphi, \mathcal{H}, \texttt{let}\ \overline{x} = \overline{n} : (\overline{\tau}[\overline{n}/\overline{x}])\ \texttt{in}\ (\tau[\overline{n}/\overline{x}])e)}$$

S-STRWIDEN

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

S-IFNTT

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

Fig. 5: CORECHKC 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 }
```

Fig. 6: Implementation of safe `strcat`

uation 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 CORECHKC, but will violate Theorem 4, which we introduce in Sec. IV.

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 $\texttt{let}\ x = \texttt{strlen}(y)\ \texttt{in}\ e$. Lines 4 and 5 in Fig. 6 are examples. The widened upper bound precipitated by $\texttt{strlen}(y)$ 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 runtime as long as we are within the stack frame, and we show this does not necessarily require the use of fat pointers in Sec. IV.

Rule S-STRWIDEN implements `strlen` widening. The predicate $\forall i. n \le i < n + n_a \Rightarrow (\exists n_i\ t_i. \mathcal{H}(n + i) = n_i : \tau_i \wedge n_i \neq 0))$ 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 $n_a \le n_h$ case is handled by a normal `strlen` rule; see the supplemental report [15].)

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. 2; thus, line 11 is valid because `s`'s upper bound is properly extended.

C. Typing

We now turn to the CORECHKC 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. 7. In the rules, $m \leq m'$ uses the two-point lattice with $\mathtt{u} < \mathtt{c}$. All remaining rules are given in the supplemental report [15].

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 $m = \mathtt{u}$ 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 $m = \mathtt{c}$), but τ is unrestricted when $m = \mathtt{u}$. The T-CASTCHECKEDPTR rule permits casting from an expression of type τ' to a checked pointer when $\tau' \sqsubseteq \mathtt{ptr}^{\mathtt{c}}\ \tau$. This subtyping relation $\sqsubseteq$ is given in Fig. 8; the many rules ensure the relation is transitive. Most of the rules handle casting between array pointer types. The second rule $0 \leq b_l \wedge b_h \leq 1 \Rightarrow \mathtt{ptr}^m\ \tau \sqsubseteq \mathtt{ptr}^m\ [(b_l, b_h)\ \tau]$ 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.[5] Θ is a map from variables x to predicates P, which have the form $P ::= \top \mid \mathtt{ge\_0}$. 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.[6]

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 $\mathtt{strlen}(y)$ 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 $\mathtt{strlen}(y)$; 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. III-D shows that our type system is a sound model of the CORECHKC semantics, and we discuss how we guarantee that the behavior of our compiler formalization and the semantics matches in Sec. IV.

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. 6 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 $\mathtt{x} \geq 0$. The static cast on line 14 is valid according to the last line in Fig. 8:

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

because $0 \leq 0$ and $0 \leq x$, where the latter holds since Θ proves $x \geq 0$. 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 [22, 26].

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. 6; 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. 3).

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

[6] 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 the supplemental report [15].

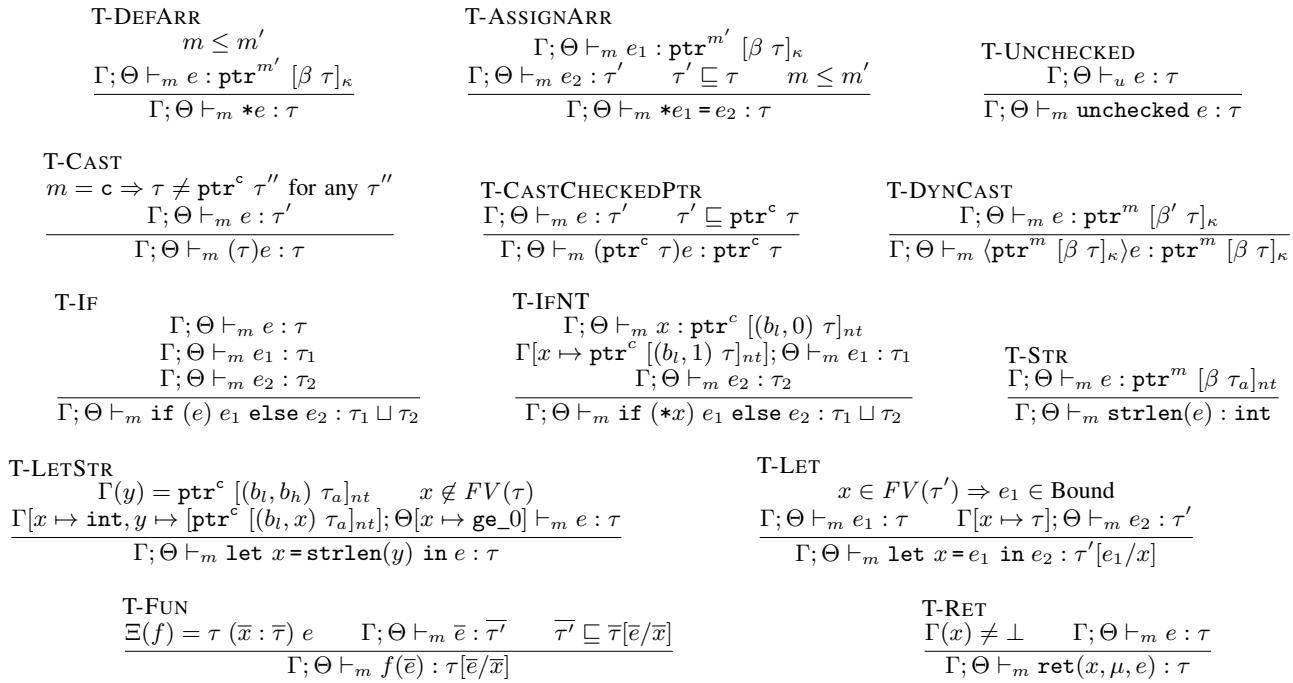

Fig. 7: Selected type rules

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

Fig. 8: Subtyping

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. 5); 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 $n{:}\tau$.

D. Type Soundness and Blame

In this subsection, we focus on our main meta-theoretic results about CoreChkC: 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. [23], and show the rules in Fig. 9; the judgment $\mathcal{H};\sigma \vdash n : \tau$ is used to confirm

Type rules for constants and variables:

$$\text{T-VAR}\ \frac{x : \tau \in \Gamma}{\Gamma; \Theta \vdash_m x : \tau} \qquad \text{T-CONST}\ \frac{FV(\tau) = \emptyset \qquad \mathcal{H}; \emptyset \vdash n : \tau}{\Gamma; \Theta \vdash_m n{:}\tau : \tau}$$

Rules for checking constant pointers:

$$\mathcal{H}; \sigma \vdash n : \texttt{int} \qquad \mathcal{H}; \sigma \vdash n : \texttt{ptr}^{\texttt{u}}\ \omega \qquad \mathcal{H}; \sigma \vdash 0 : \texttt{ptr}^{\texttt{c}}\ \omega$$

$$\frac{(n{:}\texttt{ptr}^{\texttt{c}}\ \omega) \in \sigma}{\mathcal{H}; \sigma \vdash n : \texttt{ptr}^{\texttt{c}}\ \omega}$$

$$\frac{\forall i \in [0, \mathit{size}(\omega)).\mathcal{H}; (\sigma \cup \{(n : \texttt{ptr}^{\texttt{c}}\ \omega)\}) \vdash \mathcal{H}(n+i)}{\mathcal{H}; \sigma \vdash n : \texttt{ptr}^{\texttt{c}}\ \omega}$$

Fig. 9: Type Rules for Constants/Variables

literal well-typing, where σ is a set of pointer literals already checked in $\mathcal{H}$ (to allow pointer cycles). See the supplemental report [15] 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 ($\Gamma; \Theta \vdash_{\texttt{c}} e : \tau$ for some τ), one of the following holds:

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

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 ($\Gamma; \Theta \vdash_{\texttt{c}} e : \tau$ for some τ), if there exists φ', $\mathcal{H}'$ and e', such that $(\varphi, \mathcal{H}, e) \longrightarrow_{\texttt{c}} (\varphi', \mathcal{H}', e')$, 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 ($\Gamma'; \Theta' \vdash_{\texttt{c}} e : \tau'$), 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[7]—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 ($\varphi; \Theta \vdash_{\texttt{c}} e : \tau$ 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 $m_j = \texttt{u}$, or there exists E and e', such that $r = E[e']$ and $\mathit{mode}(E) = \texttt{u}$.

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. [23], proofs for CoreChkC 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).

IV. Compilation

The semantics of CoreChkC 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 CoreChkC 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 CoreChkC to CoreC, an untyped language without metadata annotations. The syntax and semantics CoreC closely mirrors that of CoreChkC; it differs only in that literals lack type annotations and its operational rules perform no bounds and null checks, which are instead inserted during compilation. Our compilation algorithm is evidence that CoreChkC'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 CoreChkC'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.[8]

We formalize rules for this judgment in PLT Redex [6], following and extending our Coq development for CoreChkC. 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.

[7]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.

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

```
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 }
```

Fig. 10: Compilation Example for Check Insertions

A. Approach

Due to space constraints, we explain the rules for compilation by example, using a C-like syntax; the complete rules are given in the supplemental report [15]. 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.[9] Fig. 10 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. 5), 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 `p` is in bounds before computing `strlen(p)`, while line 10 does likewise before computing `*(p+1)`.

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

```
1  int deref_array(n : int,
2      p : ptr^c [(0,n) int]_nt) {
3    /* ρ(p) = p_lo,p_hi */
4    if (* p)
5      (* (p + 1))
6      else 0
7  }
8  ...
9  /* p0 : ptr^c [(0,5) int]_nt */
10 deref_array(5, p0);
```

```
1  deref_array(n, p) {
2    let p_lo = 0;
3    let p_hi = n;
4    /* runtime checks */
5    assert(p_lo ≤0 && 0 ≤p_hi);
6    assert(p != 0);
7    let p_derefed = *p;
8    if (p_derefed != 0) {
9      /* widening */
10     if (p_hi == 0) {
11       p_hi = p_hi + 1;
12     }
13     /* null check before pointer arithmetic */
14     assert(p != 0);
15     let p0 = p + 1;
16     assert(p_lo ≤1 && 1 ≤p_hi);
17     (* p0)
18   }
19   else {
20     0
21   }
22 }
23 ...
24 deref_array(5, p0);
```

Fig. 11: Compilation Example for Dependent Functions

For `strlen(p)` and conditionals `if(*p)`, the CoreChkC 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. 10 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. 11 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. 11 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 CoreChkC 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`.

B. 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

```
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 }
```

Fig. 12: Safe `strcat` in Checked C that avoids a run-time error exhibited by `safe_strcat` (Fig. 6) when compiled with the current Checked C compiler

specification and current implementation of Checked C [25, 5.1.2, pg 85]. For example, the `safe_strcat` example of Fig. 6 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 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. 12 (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. 10, 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.

C. Metatheory

We formalize both the compilation procedure and the simulation theorem in the PLT Redex model we developed for CoreChkC (see Sec. III-A), 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 CoreChkC, the semantics of CoreC does not distinguish checked and unchecked regions.

Fig. 13 gives an overview of the simulation theorem.[10] The simulation theorem is specified in a way that is similar to the one by Merigoux et al. [18]. An ordinary simulation 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 CoreChkC 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 $(\varphi_0, \mathcal{H}_0, e_0) \rightarrow_{\mathsf{c}} (\varphi_1, \mathcal{H}_1, r_1)$, 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 $r_1 = \texttt{bounds}$ or $\texttt{null}$, 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

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

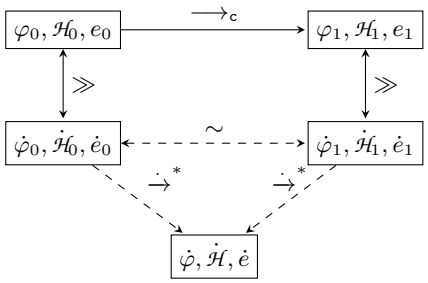

Fig. 13: Simulation between CoreChkC and CoreC

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.

V. Random Testing via the Implementation

In addition to using the CoreChkC Redex model to establish simulation of compilation (Section IV-C), 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.

Generating Well Typed Terms. For this random generation, we follow the approach of Pałka et al. [21] 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[11], color-coded to represent inputs and outputs of the generation process:[12]

$$\text{G-DefArr}\quad \frac{\Gamma; \Theta \vdash_m e : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad m \leq m'}{\Gamma; \Theta \vdash_m *e : \tau}$$

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 $\Gamma; \Theta \vdash_m e : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa$, 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 $\texttt{u}$, then m' needs to be $\texttt{u}$ 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 [12]. 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. [21] 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 $(0, 0)$.

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

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. [21], 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.

[11] Generator rules G-* correspond one to one with the type rules T-* in Sec. III-C.

[12] 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.

Generating Ill-typed Terms. We can use generated well-typed terms to test our simulation theorem (Section IV) and test that CoreChkC and Checked C Clang agree on what is type-correct. But it is also useful to generate ill-typed terms to test that CoreChkC 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 CoreChkC 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. [8], 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:

```
array_ptr<char> fun(void) : count(3) {
  array_ptr<char> x : count(3);
  x = calloc(3, sizeof(char));
  return x+3;
}
int main(void) {
  *(fun()) = 0;
  return 0;
}
```

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:[13]

```
array_ptr<int> f(void) : count(5) {
  array_ptr<int> x : count(5) =
    calloc<int>(5, sizeof(int));
  return x;
}
array_ptr<int> g(void ) : count(5) {
  array_ptr<int> x : count(5) =
    calloc<int>(5, sizeof(int));
  return x+3;
}
int main(void) {
  return *(0 ? g() : f() + 3);
}
```

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.

VI. 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, 14], Ellison and Rosu [5], Kang et al. [11], and Memarian et al. [16, 17]. 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 CoreChkC, 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. [9] extends the formalization of Ellison and Rosu [5] 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 CoreChkC's. Its concern is bug finding, not compiling programs to use this semantics.

CCured [20] and Softbound [19] implement spatially safe semantics for normal C via program transformation. Like

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

CoreChkC, 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 IV. 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 [7, 10] is a C dialect that aims to ensure memory safety; its pointer types are similar to CCured. Cyclone's formalization [7] focuses on the use of *regions* to ensure temporal safety; it does not formalize arrays or threats to spatial safety. Deputy [2, 29] is another safe-C dialect that aims to avoid fat pointers; it was an initial inspiration for Checked C's design [4], though it provides no specific modeling for null-terminated array pointers. Deputy's formalization [2] defines its semantics directly in terms of compilation, similar in style to what we present in Section IV. Doing so tightly couples typing, compilation, and semantics, which are treated independently in CoreChkC. 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. [23]. 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. CoreChkC's Coq-based development (Section III) 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 CoreChkC 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.

VII. Conclusion and Future Work

This paper presented CoreChkC, a formalization of an extended core of the Checked C language which aims to provide spatial memory safety. CoreChkC 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 CoreChkC (whose semantics leverage fat pointers) can be compiled to CoreC (which does not) while preserving their behavior. We developed a version of CoreChkC 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 CoreChkC against the Checked C compiler, finding multiple inconsistencies in the process.

As future work, we wish to extend CoreChkC 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 [13] or VeLLVM [28], with Checked C features, towards producing a verified-correct Checked C compiler. We believe that CoreChkC's Coq and Redex models lay the foundation for such a step, but substantial engineering work remains.

Acknowledgments: We thank the anonymous reviewers for their helpful, constructive comments. This work was supported in part by a gift from Microsoft.

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] 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.

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Liyi Li, Yiyun Liu, Deena Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. A Formal Model of Checked C (extended version). https://arxiv.org/abs/2201.13394, 2022.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

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

[26] 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.

[27] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A Retargetable Framework for Low-level Inlined-reference Monitors. In *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

[28] 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.

[29] 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.