Into the Depths of C: Elaborating the De Facto Standards

Kayvan Memarian[1] Justus Matthiesen[1] James Lingard[2] Kyndylan Nienhuis[1]
David Chisnall[1] Robert N.M. Watson[1] Peter Sewell[1]

[1]University of Cambridge, UK [2]University of Cambridge, UK (when this work was done)

[1]`first.last@cl.cam.ac.uk` [2]`james@lingard.com`

Abstract

C remains central to our computing infrastructure. It is notionally defined by ISO standards, but in reality the properties of C assumed by systems code and those implemented by compilers have diverged, both from the ISO standards and from each other, and none of these are clearly understood.

We make two contributions to help improve this error-prone situation. First, we describe an in-depth analysis of the design space for the semantics of pointers and memory in C as it is used in practice. We articulate many specific questions, build a suite of semantic test cases, gather experimental data from multiple implementations, and survey what C experts believe about the de facto standards. We identify questions where there is a consensus (either following ISO or differing) and where there are conflicts. We apply all this to an experimental C implemented above capability hardware. Second, we describe a formal model, Cerberus, for large parts of C. Cerberus is parameterised on its memory model; it is linkable either with a candidate de facto memory object model, under construction, or with an operational C11 concurrency model; it is defined by elaboration to a much simpler Core language for accessibility, and it is executable as a test oracle on small examples.

This should provide a solid basis for discussion of what mainstream C is now: what programmers and analysis tools can assume and what compilers aim to implement. Ultimately we hope it will be a step towards clear, consistent, and accepted semantics for the various use-cases of C.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]

Keywords C

1. Introduction

C, originally developed 40+ years ago, remains one of the central abstractions of our computing infrastructure, widely used for systems programming. It is notionally defined by ANSI/ISO standards, C89/C90, C99 and C11, and a number of research projects have worked to formalise aspects of these [3, 16, 18, 25–29, 38–40]. But the question of what C is in reality is much more complex, and more problematic, than the existence of an established standard would suggest; formalisation alone is not enough.

Problem 1: the de facto standards vs the ISO standard In practice, we have to consider the behaviours of the various mainstream C compilers, the assumptions that systems programmers make about the behaviour they can rely on, the assumptions implicit in the corpus of existing C code, and the assumptions implicit in C analysis tools. Each of these *de facto* standards is itself unclear and they all differ, from each other and from the ISO standard, despite the latter's stated intent to *"codify the common, existing definition of C"*.

This is not just a theoretical concern. Over time, C implementations have tended to become more aggressively optimising, exploiting situations which the standards now regard as undefined behaviour. This can break code that used to work correctly in practice, sometimes with security implications [47, 48]. As we shall see, critical infrastructure code, including the Linux and FreeBSD kernels, depends on idioms that current compilers do not in general support, or that the ISO standard does not permit. This leads to tensions between the compiler and OS-developer communities, with conflicting opinions about what C implementations do or should guarantee. One often sees large projects built using compiler flags, such as `-fno-strict-aliasing` and `-fno-strict-overflow`, that turn off particular analyses; the ISO standard does not describe these.

Meanwhile, developers of static and dynamic analysis tools must make ad hoc semantic choices to avoid too many "false" positives that are contrary to their customers' expectations, even where these are real violations of the ISO standards [5]. This knowledge of the de facto standards is often left implicit in the tool implementations, and making it precise requires an articulation of the possible choices.

At the same time, in other respects the space of mainstream C implementations has become simpler than the one that the C standard was originally written to cope with. For example, mainstream hardware can now reasonably be assumed to have 8-bit bytes, twos-complement arithmetic, and (often) non-segmented memory, but the ISO standard does not take any of this into account.

Problem 2: no precise and accessible specification Focussing on the ISO standard alone, a second real difficulty is that it is hard to interpret precisely. C tries to solve a challenging problem: to simultaneously enable programmers to write high-performance hand-crafted low-level code, provide portability between widely varying target machine architectures, and support sophisticated compiler optimisations. It does so by providing operations both on abstract values and on their underlying concrete representations. The interaction between these is delicate, as are the dynamic properties relating to memory and type safety, aliasing, concurrency, and so on; they have, unsurprisingly, proved difficult to characterise precisely in the prose specification style of the standards. Even those few who are very familiar with the standard often struggle with the subtleties of C, as witnessed by the committee's internal mailing list discussion of what the correct interpretation of the standard should be, the number of requests for clarification made in the form of defect reports [51], and inconclusive discussions of whether compiler anomalies are bugs w.r.t. the standard. The prose standard is not executable as a test oracle, which would let one simply compute the set of all allowed behaviours of any small test case, so such discussions have to rely on exegesis of the standard text. The goal of ANSI C89, to *"develop a clear, consistent, and unambiguous Standard for the C programming language"* [2], thus remains an open problem.

The obvious semanticist response is to attempt a mathematical reformulation of the standard, as in the above-cited works, but this ignores the realities of the de facto standards. Moreover, a language standard, to serve as the contract between implementors and users, must be accessible to both. Mathematical definitions of semantics are precise, especially if mechanised, but (especially mechanised) are typically not accessible to practitioners or the C standards community.

Problem 3: integrating C11 concurrent and sequential semantics Finally, there is the technical challenge of dealing with concurrency. The C standard finally addressed concurrency in C11, following C++ [1, 4, 8], and Batty et al. [3] worked to ensure that here, unusually, the standard text is written in close correspondence to a formalisation. But that concurrency model is in an axiomatic style, while the rest of the standard is more naturally expressed in an operational semantics; it is hard to integrate the two.

To summarise, the current state of C is, simply put, a mess. The divergence among the de facto and ISO standards, the prose form, ambiguities, and complexities of the latter, and the lack of an integrated treatment of concurrency, all mean that the ISO standard is not at present providing a satisfactory definition of C as it is or should be. It also does not provide a good basis for designing future refinements of C. The scale of the problem, and the real disagreements about what C is, mean that there is no simple solution to all this, but we take several steps to clarify the situation.

Contributions Our first contribution is a detailed investigation into the possible semantics for pointers and memory in C (its *memory object model*), where there seems to be the most divergence and variation among the multiple de facto and ISO standards. We explore this in four interlinked ways:

1. An in-depth analysis of the design space. We identify 85 questions, supported by 196 hand-written semantic test cases, and for each discuss the desired behaviour w.r.t. the ISO standard and real-world implementation and usage.
2. A survey investigating the de facto standards, directly probing what systems programmers and compiler writers believe about compiler behaviour and extant code.
3. Experimental data for our test suite, for GCC and Clang (for multiple versions and flags), the Clang undefined behavior, memory, and address sanitisers, TrustInSoft's `tis-interpreter` [44], and KCC [16, 18, 20].
4. In ongoing work, we are building a candidate formal model capturing one plausible view of the de facto standards, as a reference for discussion.

These feed into each other: the survey responses and experimental data inform our formal modelling, and the design of the latter, which eventually must provide a single coherent semantics that takes a position on the allowed semantics of arbitrary code, has raised many of the questions.

Our focus here is on current mainstream C, not on the C of obsolete or hypothetical implementations. We aim to establish a solid basis for future discussion of the de facto standards, identifying places where there is a more-or-less clear consensus (either following ISO or differing) and places where there are clear conflicts.

This has already proved useful in practice: we applied our analysis and test suite to the C dialect supported by Watson et al's experimental CHERI processor [49, 50], which implements unforgeable, bounds-checked C pointers (capabilities). CHERI C's strong dynamically enforced memory safety is expected to be more restrictive than mainstream C, but also to support existing systems software with only modest adaptation. We identified several hardware and software bugs and important design questions for CHERI C, building on and informing our analysis of the de facto standards.

For the memory object model, the de facto standard usage and behaviour are especially problematic, but many of the other subtle aspects of C have not been significantly affected by the introduction of abstract notions (of pointer, unspecified value, etc.); for these, while there are many ambiguities in the ISO standard text and divergences between it and practice [5, 43, 47], the text is a reasonable starting point.

Our second contribution is a formal semantics, Cerberus, for large parts of C, in which we aim to capture the ISO text for these aspects as clearly as possible. Cerberus is parameterised on the memory model, so it can be instantiated with the candidate formal model that we are developing or, in future, other alternative memory object models. It is executable as a test oracle, to explore all behaviours or single paths of test programs; this is what lets us run our test suite against the model. Cerberus can also be instantiated with an operational C/C++11 concurrency model [37], though not yet combined with a full memory object model.

Cerberus aims to cover essentially all the material of Sections 5 *Environment* and 6 *Language* of the ISO C11 standard, both syntax and semantics, except: preprocessor features, C11 character-set features, floating-point and complex types (beyond simple float constants), user-defined variadic functions (we do cover `printf`), bitfields, `volatile`, `restrict`, generic selection, `register`, flexible array members, some exotic initialisation forms, signals, `longjmp`, multiple translation units, and aspects where our candidate de facto memory object model intentionally differs from the standard. It supports only small parts of the standard libraries. Threads, atomic types, and atomic operations are supported only with a more restricted memory object model. For everything in scope, we aim to capture all the permitted semantic looseness, not just that of one or some implementation choices.

To manage some of the complexity, the Cerberus computation semantics is expressed by an elaboration (a compositional translation) from a fully type-annotated C AST to a carefully defined Core: a typed call-by-value calculus with constructs to model certain aspects of the C dynamic semantics. Together these handle subtleties such as C evaluation order, integer promotions, and the associated implementation-defined and undefined behaviour, for which the ISO standard is (largely) clear and uncontroversial. The elaboration closely follows the ISO standard text, allowing the two to be clearly related for accessibility.

The Cerberus front-end comprises a clean-slate C parser (closely following the grammar of the standard), desugaring phase, and type checker. This required substantial engineering, but it lets us avoid building in semantic choices about C that are implicit in the transformations from C source to AST done by compiler or CIL [35] front-ends.

We also report on a preliminary experiment in translation validation (in Coq) for the front-end of Clang, for very simple programs.

All this is a considerable body of material: our design-space analysis alone is an 80+ page document; we refer to the extensive supplementary material [32] for that and for our test suite, survey results, and experimental data. We summarise selected design-space analysis in §2, apply it in §3 and §4, and summarise aspects of Cerberus in §5. We discuss validation and the current limitations of the model in §6, and related work in §7.

2. Pointer and Memory Disagreements

"Why do you have to ask questions that make me want to audit all my C code?"

The most important de facto standards for C are those implicit in the billions of lines of extant code: the properties of the language implementations that all that code relies on to work correctly (to the extent that it does, of course). On the other side, we have the emergent behaviour that mainstream C compilers can exhibit, with their hundreds of analysis and optimisation passes. Both are hard to investigate directly, but usefully modelling C depends on the ability to define these real-world variants of C. Our surveys are, to the best of our knowledge, a novel approach to investigating the de facto semantics of a widely used language. We produced two. The first version, in early 2013, had 42 questions, with concrete code examples and subquestions about the de facto and ISO standards. We targeted this at a small number of experts, including multiple contributors to the ISO C or C++ standards committees, C analysis tool developers, experts in C formal semantics, compiler writers, and systems programmers. The results were very instructive, but this survey demanded a lot from the respondents; it was best done by discussing the questions with them in person over several hours.

Our second version (in early 2015), was simplified, making it feasible to collect responses from a wider community. We designed 15 questions, selecting some of the most interesting issues from our earlier survey, asked only about the de facto standard (typically asking whether some idiom would work in normal C compilers and whether it was used in practice), omitted the concrete code examples, and polished the questions to prevent misunderstandings that we saw in early trials. We refer to these questions as [n/15] below. Aiming for a modest-scale but technically expert audience, we distributed the survey among our local systems research group, at EuroLLVM 2015, via technical mailing lists: gcc, llvmdev, cfe-dev, libc-alpha, xorg, freebsd-developers, xen-devel, and Google C user and compiler lists, and via John Regehr's blog, widely read by C experts. There were 323 responses, including around 100 printed pages of textual comments (the above quote among them). Most respondents reported expertise in C systems programming and many reported expertise in compiler internals and in the C standard:

C applications programming	255
C systems programming	230
Linux developer	160
Other OS developer	111
C embedded systems programming	135
C standard	70
C or C++ standards committee member	8
Compiler internals	64
GCC developer	15
Clang developer	26
Other C compiler developer	22
Program analysis tools	44
Formal semantics	18
no response	6
other	18

We also used quantitative data on the occurrence of particular idioms in systems C code from CHERI [11].

Our full set of 85 questions [10] addresses all the C memory object model semantic issues that we are currently aware of, including all those in these two surveys. We refer to these as **Qnn** below; they can be categorised as follows (with the number of questions in each category):

Pointer provenance basics	3
Pointer provenance via integer types	5
Pointers involving multiple provenances	5
Pointer provenance via pointer representation copying	4
Pointer provenance and union type punning	2
Pointer provenance via IO	1
Stability of pointer values	1
Pointer equality comparison (with == or !=)	3
Pointer relational comparison (with <, >, <=, or >=)	3
Null pointers	3
Pointer arithmetic	6
Casts between pointer types	2
Accesses to related structure and union types	4
Pointer lifetime end	2
Invalid accesses	2
Trap representations	2
Unspecified values	11
Structure and union padding	13
Basic effective types	2
Effective types and character arrays	1
Effective types and subobjects	6
Other questions	5

In this section we discuss some of the key points of disagreement between practitioners, implementers and the standard with regard to pointers and memory, referring to our survey and experimental results. Of the 85 questions,

- for 39 the ISO standard is unclear;
- for 27 the de facto standards are unclear, in some cases with significant differences between usage and implementation; and
- for 27 there are significant differences between the ISO and the de facto standards.

2.1 Pointer Provenance

Originally one could think of C as manipulating *"the same sort of objects that most computers do, namely characters, numbers, and addresses"*, Kernigan and Ritchie [24, p.2]. At runtime, for conventional C implementations, that is still basically true, but the current ISO standards involve more abstract values, for pointers, unspecified values, and typed regions of memory; they cannot be considered as simple bit-vector-represented quantities. Compile-time analyses rely on the more abstract notions to legitimise optimisation transforms, and exactly what they are leads to some of the most important and subtle questions about C.

The ISO WG14 Defect Report DR260 Committee Response [53] declares that *"The implementation is entitled to take account of the provenance of a pointer value when determining what actions are and are not defined."*. This is observable in practice for the following example from our test suite, adapted from DR260.

EXAMPLE (`provenance_basic_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
int  y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
  return 0;
}
```

If `x` and `y` happen to be allocated in adjacent memory, `&x+1` and `&y` will have bitwise-identical runtime representation values, the `memcmp` will succeed, and `p` (derived from a pointer to `x`) will have the same representation value as the pointer to `y` (a different object) at the point of the update `*p=11`. In a concrete semantics we would expect to see `x=1 y=11 *p=11 *q=11`, but GCC produces `x=1 y=2 *p=11 *q=2` (ICC produces `x=1 y=2 *p=11 *q=11`). This suggests that GCC is reasoning, from provenance information, that `*p` does not alias with `y` or `*q`, and hence that the initial value of `y=2` can be propagated to the final `printf`. Note that this is not a type-based aliasing issue: the pointers are of the same type, and the GCC result is not affected by `-fno-strict-aliasing`.

DR260 suggests a semantics in which pointer values include not just a concrete address but also provenance information, erased at runtime in conventional implementations, including a unique ID from the original allocation. That can be used in the semantics for memory accesses to check that the address used is consistent with the original allocation, which here lets the `*p = 11` access be regarded as *undefined behaviour*. The existence of an execution with undefined behaviour means this program is considered erroneous and compilers are notionally entirely unconstrained in how they treat it — thus making the analysis and optimisation (vacuously) correct in this case. This general pattern is typical for C: regarding situations as undefined behaviour puts an obligation on programmers to avoid them, but permits compilers to make strong assumptions when optimising.

So far this is uncontroversial among C experts, though it may be surprising to anyone at first sight, and several models for C or for particular C-like languages have had semantics for pointers with some kind of (block-ID, offset) model. But it leads to many more vexed questions, of which we give a sample below.

Q25 Can one do relational comparison (with <, >, <=, or >=) of two pointers to separately allocated objects (of compatible object types)? ISO clearly prohibits this [1, §6.5.8p5], but our surveys show that it is widely used, e.g. for global lock orderings and for collection-implementation orderings. Numerically (survey question [7/15]), we get: *Will that work in normal C compilers?* yes:

191 (60%) only sometimes: 52 (16%), no: 31 (9%), don't know: 38 (12%), I don't know what the question is asking: 3 (1%), and *Do you know of real code that relies on it?* yes: 101 (33%), yes, but it shouldn't: 37 (12%), no, but there might well be: 89 (29%), no, that would be crazy: 50 (16%), don't know: 27 (8%).

The only cases where it is widely thought to fail are for segmented architectures, but those are now quite uncommon (e.g. AS/400 and old x86 processors) except for some mainframe architectures. For mainstream C semantics, it seems more useful to permit it than to take the strict-ISO view that all occurrences are bugs. (One respondent encountered problems for pointers spanning the middle of the address space). In our model, we can easily regard these relational comparisons as ignoring the provenance information.

In real C this is one of several ways in which concrete address values are exposed to programs (along with explicit casts to integer types, IO of pointers, and examination of pointer representation bytes). Abstract pointer values must also therefore contain concrete addresses, in contrast to those earlier C semantics that only had block IDs, and the semantics must let those be nondeterministically chosen in any way that a reasonable implementation might.

Q9 Can one make a usable offset between two separately allocated objects by inter-object integer or pointer subtraction, making a usable pointer to the second by adding the offset to a pointer to the first? Here again we see a conflict, now with C usage on one side and compilers and ISO on the other. In practice, this usage is uncommon, but the basic pattern does occur in important specific cases, e.g. in the Linux and FreeBSD implementations of per-CPU variables. However, current compilers sometimes do optimise based on an assumption, in a points-to analysis, that inter-object pointer arithmetic does not occur.

How could this be resolved? One could argue that the usages are bugs and should be rewritten, but that would needlessly lose performance; it seems unlikely to be acceptable. One could globally turn off those optimisations, e.g. with `-fno-tree-pta` for GCC, but that too is a blunt instrument. One could adapt the compiler analyses to treat inter-object pointer subtractions as giving integer offsets that have the power to move between objects. It would be easy to define a corresponding multiple-provenance semantics, with provenances that could be wildcards or sets of allocation IDs, not just the singleton provenances implicit in DR260, but implementations would have to be conservative where they could not determine that pointer subtractions are intra-object; that also might be too costly. Or, finally, one could require such usages to be explicitly annotated, e.g. with an attribute on the resulting pointer that declares that it might alias with anything. None of these are wholly satisfactory. We suspect the last is the most achievable, but for the moment our candidate formal model forbids this idiom.

As for many of our questions, there are essentially social or political questions as to whether the ISO standard, common usage, and/or mainstream compiler behaviour can be changed. That is beyond the scope of this paper; the most we can do is make a start on clearly articulated possibilities.

Q5 Must provenance information be tracked via casts to integer types and integer arithmetic? The survey shows that it is common to cast pointers to integer types and back to do arithmetic on them, e.g. to store information in unused bits. To define a coherent model we must either regard the result of casting from an integer type to a pointer as of a wildcard provenance or track provenance through integer operations. The GCC documentation states *"When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined."*, strongly suggesting the latter, but raising the question of what to do if there is no single "original pointer". Our formal model associates provenances with all integer values, following the same at-most-one provenance model we use for pointers.

Q2 Can equality testing on pointers be affected by pointer provenance information? The ISO standard explicitly permits equality comparison between pointers to separately allocated objects (of compatible types) [1, §6.5.9], but leaves open whether the result of such comparison might depend on provenance (presumably simply because the text has not been systematically revised since DR260). Experimentally, our test suite includes cases where GCC regards two pointers with the same runtime representation but different provenances as unequal if the allocations and comparison are in the same compilation unit but equal if they are split across two compilation units, showing that these compilers do take advantage of the provenance information if it is statically available. This is not a semantics that we would a priori choose as language designers, but it can be soundly modelled by making a nondeterministic choice at each such comparison whether to take provenance into account or not. Writing a formal semantics usefully forces us to consider such questions; it also reveals the language-definition-complexity cost of such optimisations.

2.2 Out-of-Bounds Pointers

The ISO standard permits only very limited pointer arithmetic, essentially within an array, among the members of a struct, to access representation bytes, or one-past an object. But in practice it seems to be common to transiently construct out-of-bounds pointers (**Q31**). Chisnall et al. found this in 7 of the 13 C codebases they examined [11, Table 1]. So long as they are brought back in-bounds before being used to access memory, many experts believe this will work; our survey (Question [9/15]) gave: yes: 230 (73%), only sometimes: 43 (13%), no: 13 (4%), don't know: 27 (8%). On the other side, the textual comments reveal that some compiler developers believe that compilers may optimise assuming that this does not occur, and there can be issues with overflow in pointer arithmetic and with large allocations.

This is another clear conflict. For our formal model, as in CHERI, we tentatively choose to permit arbitrary pointer arithmetic: out-of-bounds pointers can be constructed, with undefined behaviour occurring if an access-time check w.r.t. the bounds associated to their provenance fails.

2.3 Pointer Copying

In a provenance semantics, as C lets one operate on the representation bytes of values, we have to ask when those operations make usable copies of pointers, preserving the original provenance. The library `memcpy` must clearly allow this, but what about user code that copies the representation bytes, perhaps with more elaborate computation on the way (**Q13–Q16**)? Most survey respondents expect this to work (Question [5/15]): yes: 216 (68%), only sometimes: 50 (15%), no: 18 (5%), don't know: 24 (7%). They give some interesting examples, e.g. *"Windows /GS stack cookies do this all the time to protect the return address. The return address is encrypted on the stack, and decrypted as part of the function epilogue"* and a *"JIT that stores 64-bit virtual ptrs as their hardware based 48-bits"*. Our candidate formal model should permit copying pointer values via indirect dataflow, as the representation bytes or bits carry the original provenance, combining values with the same provenance preserves that provenance, and the access-time check compares the recalculated address with that of the original allocation. It will not permit copying via indirect control flow (e.g. making a branch based on each bit of a pointer value), and it intentionally does not require all of the original bits to flow to the result. We view this as a plausible de facto semantics, but more work is needed to see if it is really compatible with current compiler analysis implementations.

2.4 Unspecified Values

C does not require variables and memory to be initialised. Reading an uninitialised variable or struct member (either due to a bug or intentionally, to copy, output, hash, or set some bits of a partially initialised value), has several possible semantics. Our survey (Question [2/15]) gave bimodal answers, split between (1) and (4):

1. undefined behaviour (meaning that the compiler is free to arbitrarily miscompile the program, with or without a warning): 139 (43%)
2. going to make the result of any expression involving that value unpredictable: 42 (13%)
3. going to give an arbitrary and unstable value (maybe with a different value if you read again): 21 (6%)
4. going to give an arbitrary but stable value (with the same value if you read again): 112 (35%)

In the text responses, the only real use cases seem to be copying a partially initialised struct and (more rarely) comparing against one. It appears that current Clang, GCC, and MSVC are not exploiting the licence of (1), though one respondent said Clang is moving towards it, and one that (1) may be required for Itanium. Another makes a strong argument that the MSVC behaviour is more desirable for security reasons. But GCC and Clang do perform SSA transformations that make uninitialised values unstable (2); our test cases exhibit this for Clang.

The ISO standard introduces *indeterminate values*, which are either an *unspecified value* (a *"valid value [where ISO] imposes no requirements on which value is chosen in any instance"* or a *trap representation*; reading uninitialised values can give undefined behaviour either if the type being read has trap representations in this implementation, or (6.3.2.1p2) if the object has not had its address taken. These definitions have given rise to much confusion over the years, but it seems clear that for current mainstream C, there are no trap representations at most types, perhaps excepting `_Bool`, floating-point, and some mainframe pointer types, and the 6.3.2.1p2 clause was intended to cover the Itanium case. Leaving those cases apart, (2) seems reasonable from the compiler point of view, in tension with (4), that may be relied upon by some code.

2.5 Unspecified Padding Values

Unspecified values arise also in padding bytes, which in C can be inspected and mutated via `char *` pointers. Here we see several possible semantics, including:

1. Padding bytes are regarded as always holding unspecified values, irrespective of any byte writes to them (so the compiler could arbitrarily write to padding at any point).
2. Structure member writes are deemed to also write unspecified values over subsequent padding.
3. ...or to nondeterministically either write zeros over subsequent padding or leave it unchanged.
4. Structure copies might copy padding, but structure member writes never touch padding.

Our survey (Question [1/15]) produced mixed results: sometimes it is necessary to provide a mechanism that programmers can use to ensure that no security-relevant information is leaked via padding, e.g. via (3) or (4); those also let users maintain the property that padding is zeroed, enabling deterministic bytewise CAS, comparison, marshalling, etc.. An MSVC respondent suggested it provides (4), and a Clang respondent that it does not require (1) or (2). But a GCC respondent suggested a plausible optimisation, scalar replacement of aggregates, which could require (1) or (2) to make the existing compiler behaviour admissible, and we understand that an IBM mainframe compiler may by default also require those. This is a real conflict.

2.6 Effective Types

C99 introduced *effective types* to permit compilers to do optimisations driven by type-based alias analysis (TBAA), ruling out programs involving unannotated aliasing of references to different types by regarding them as having undefined behaviour. This is one of the less clear, less well-understood, and more controversial aspects of C. The effective-types question of our preliminary survey was the only one which received a unanimous response: "don't know". Here again we find conflicts, for example for:

Q75 Can an unsigned character array with static or automatic storage duration be used (in the same way as a `malloc`'d region) to hold values of other types?

In our survey (Question [11/15]), 243 (76%) say this will work, and 201 (65%) know of real code that relies on it, but a strict reading of the ISO standard disallows it, and a GCC contributor noted *"No, this is not safe (if it's visible to the compiler that the memory in question has unsigned char as its declared type"*. However, our candidate formal model focusses on the C used by systems code, often compiled with `-fno-strict-aliasing` to turn TBAA off, and so should permit this and related idioms.

3. Memory Semantics of C Analysis Tools

As an initial experimental investigation into this, we ran our tests with Clang's memory, address, and undefined-behavior sanitisers (MSan, ASan, and UBSan), which are intended to identify uses of uninitialized memory values, invalid memory accesses, and other undefined behaviors, the TrustInSoft `tis-interpreter` [44] (based on the Frama-C value analysis [9]), and Hathhorn et al.'s KCC [16, 18, 20]. We discussed the `tis-interpreter` results briefly with one of its authors. These three groups of tools gave radically different results.

For the Clang sanitisers, we were surprised at how few of our tests triggered warnings. All 13 of our structure-padding tests and 9 of our other unspecified-value tests ran without any sanitiser warnings (including tests that triggered compile-time warnings). For example, **Q49** passes an unspecified value to a library function, yet does not trigger a report, though MSan does detect a flow-control choice originating from an unspecified value in **Q50**. MSan flagged two of the unspecified value tests (though only at `-O0`). ASan and MSan did report errors on the two tests that rely on treating an arbitrary integer value as a pointer (though these also caused segmentation faults without the sanitisers enabled), but neither flagged any of our other pointer provenance tests as dubious. This might be due to deliberate design choices to adopt a liberal semantics to accommodate the de facto standards, or to limitations in the tools, or both.

`tis-interpreter` aims for a tight semantics, to detect enough cases to ensure *"that any program that executes correctly in `tis-interpreter` behaves deterministically when compiled and produces the same results"*. In many places it follows a much stricter notion of C than our candidate de facto model, e.g. flagging most of the unspecified-value tests, and not permitting comparison of pointer representations; in some others it coincides with our candidate de facto model but differs from ISO, e.g. assuming null pointer representations are zero. Our tests also identified two bugs in the tool, acknowledged and fixed by the developers.

KCC detected two potential alignment errors in earlier versions of our tests. But it gave 'Execution failed', with no further details, for the tests of 20 of our questions; 'Translation failed' for one; segfaulted at runtime for one; and gave results contrary to our reading of the ISO standard for at least 6: it exhibited a very strict semantics for reading uninitialised values (but not for padding bytes), and permitted some tests that ISO effective types forbid.

4. Memory Semantics of CHERI C

A significant amount of research has focused on memory-safe implementations of C, e.g. [14, 15, 22, 34, 35], including commercial implementations aimed at mass use, such as Intel's MPX [21]. To date, no work in this area has clearly defined the interpretation of C that it aims to implement.

The CHERI processor [54] extends existing instruction sets to support spatial memory safety. We have run our tests on the CHERI C implementation (Clang-based on an FPGA soft-core CPU), with a view to specialising our candidate Cerberus de facto model to precisely characterise their intended semantics. We found several areas where the current CHERI implementation deviates from the expected behaviour. Some were known, e.g. correct provenance on pointers to globals requires linker support which is not yet completed. Others were surprising. For example, the CHERI pointer equality had two pointers with different provenance compare equal, but not be interchangeable. This was addressed by the CHERI developers adding a new compare-exactly-equal to their instruction set to compare pointers by both their address and their metadata, to use for pointer comparison. A more subtle issue was discovered in a test where `(i & 3u) == 0u` (where `i` is a `uintptr_t`) evaluated to false, even though the low three bits in `i` are all zero. This was caused by the result of `(i & 3u)` being the fat pointer `i` with its offset set to the offset of `i` anded with 3, which gives a non-zero value. This case is particularly interesting as it is triggered by an assertion in the test: the underlying idiom does work on CHERI, but defensively written code will fail. We are still working with the CHERI C developers to determine the best solution to this issue. We also helped codify the CHERI C constraints in a number of places. For example, its non-`intptr_t` integer values do not carry pointer provenance, and provenance in arithmetic expressions is only inherited from the left-hand side.

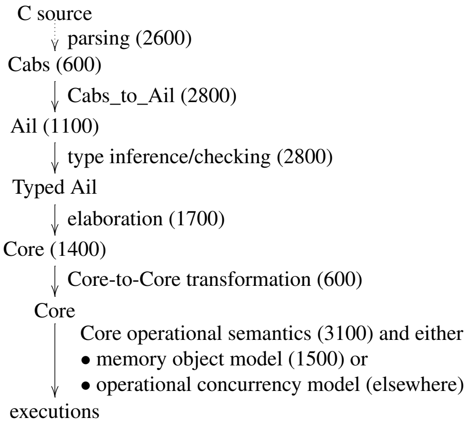

C source
parsing (2600)
Cabs (600)
Cabs_to_Ail (2800)
Ail (1100)
type inference/checking (2800)
Typed Ail
elaboration (1700)
Core (1400)
Core-to-Core transformation (600)
Core
Core operational semantics (3100) and either
• memory object model (1500) or
• operational concurrency model (elsewhere)
executions

Figure 1. Cerberus architecture (with LOS counts)

5. Cerberus

We now turn from the specifics of pointer and memory behaviour to our broader Cerberus semantics for a substantial fragment of C.

5.1 Architecture: Typed Elaboration into Core

The dynamic semantics of programming languages are often defined as operational semantics over abstract syntax types (ASTs) relatively close to the source language. Some previous work formalising large fragments of the C standard does this, e.g. Papaspyrou [40] and Ellison and Rosu [16], but it can lead to an unnecessarily monolithic semantics. Many of the dynamic intricacies of C relate to essentially compile-time phenomena, e.g. the pervasive implicit coercions for integer expressions and the loose specification of expression evaluation order. We manage complexity by factoring our semantics accordingly, into an elaboration that translates an explicitly typed source-like AST into a simpler Core language, the operational semantics of that, and either the memory object or concurrency model. Refinements to the semantics, to match some particular de facto standard, should largely affect only the first and last, leaving Core unchanged. Our elaboration function is total, and designed to produce well-typed Core programs. Both it and our Core operational semantics are as compositional as we can arrange: elaboration is inductive on the Typed AST program structure except that we precompute the lists of labels of C `case` statements; the Core operational semantics is a structural operational semantics over a state that essentially has just a Core expression and stack of continuations. By selecting an appropriate sequencing monad implementation, we can select whether to perform an exhaustive search for all allowed executions or pseudorandomly explore single execution paths.

The architecture of Cerberus is shown in Fig. 1. After conventional C preprocessing, the front end starts with a Menhir [41] parser producing values of an AST type Cabs, closely following the ISO grammar. The Cabs_to_Ail desugaring pass produces a more convenient AST, Ail. This handles many intricate aspects that might be omitted in a small calculus but have to be considered for real C, simplifying our later typechecking and elaboration. It covers: identifier scoping (linkage, storage classes, namespaces, identifier kinds); function prototypes and function definitions (including hiding, mutual recursion, etc.); normalisation of syntactic C types into canonical forms; string literals (which are implicitly allocated objects); enums (replacing them by integers and adding type annotations); and desugaring for- and do-while loops into while. Where possible, it is structured closely following the standard, and if it fails due to an ill-formed program, it identifies exactly what part of the standard is violated. Type checking adds explicit type annotations, and likewise identifies the relevant parts of the standard on any failure. The Cabs_to_Ail and typechecking passes operate without requiring any commitment to how C-standard implementation-defined choices are resolved. The main elaboration produces Core AST from Typed Ail. After optional Core-to-Core simplification, a driver combines the Core thread-local operational semantics with either our candidate de facto standard sequential memory object model or the C11 operational concurrency model.

All of this except the parser is formally specified in Lem [33] in a pure functional monadic style, to give it a clear computational intuition and permit straightforward code generation for our executable tool. It comprises around 19 000 non-comment lines of specification (LOS), plus 2600 lines of parser.

5.2 Core Overview

Core is intended to be as minimal as possible while remaining a suitable target for the elaboration, and with the behaviour of Core programs made as explicit as possible. It is a typed call-by-value language of function definitions and expressions, with first-order recursive functions, lists, tuples, booleans, mathematical integers, a type of the values of C pointers, and a type of C function designators, for function pointers. Core also includes a type `ctype` of first-class values representing C type AST terms, and various tests on it, to let Core code perform computations based on those, e.g. on whether a C type annotation in the Typed Ail source of the elaboration is a signed or unsigned integer type. The syntax of Core is in Fig. 2.

The Core type system maintains a simple distinction between pure and effectful expressions, e.g. allowing only pure expressions in the Core `if` test, and the elaboration maps as much as possible into the pure part, to ease reasoning.

C variables are mutable but Core just has identifiers that are bound to their values when introduced. Interaction with the memory object and concurrency models is factored via primitive Core *memory actions* (a) for static and dynamic object creation, kill, load, store, and read-modify-write.

To capture the intricacies of C runtime dynamics compositionally (using structural operational semantics), we add

oTy ::= **types for C objects**
$\mid$ `integer`
$\mid$ `floating`
$\mid$ `pointer`
$\mid$ `cfunction`
$\mid$ `array` (oTy)
$\mid$ `struct` tag
$\mid$ `union` tag

bTy ::= **Core base types**	
$\mid$ `unit`	unit
$\mid$ `boolean`	boolean
$\mid$ `ctype`	Core type of C type exprs
$\mid$ [bTy]	list
$\mid$ ($\overline{bTy_i}^{\,i}$)	tuple
$\mid$ oTy	C object value
$\mid$ `loaded` oTy	oTy or unspecified value

$coreTy$ ::= **Core types**	
$\mid$ bTy	pure base type
$\mid$ `eff` bTy	effectful base type

$object\_value$ ::= **C object values**	
$\mid$ $intval$	integer value
$\mid$ $floatval$	floating-point value
$\mid$ $ptrval$	pointer value
$\mid$ $name$	C function pointer
$\mid$ `array` ($\overline{object\_value_i}^{\,i}$)	C array value
$\mid$ (`struct` tag){ $\overline{.member_i = memval_i}^{\,i}$ }	C struct value
$\mid$ (`union` tag){$.member = memval$}	C union value

$value$ ::= **Core values**	
$\mid$ $object\_value$	C object value
$\mid$ `Specified` ($object\_value$)	non-unspecified loaded value
$\mid$ `Unspecified` ($ctype$)	unspecified loaded value
$\mid$ `Unit`	unit
$\mid$ `True`	true
$\mid$ `False`	false
$\mid$ $ctype$	C type expr as value
$\mid$ bTy[$value_1$, .. ,$value_n$]	list
$\mid$ ($value_1$, .. ,$value_n$)	tuple

$ptrop$ ::= **pointer operations involving the memory state**	
$\mid$ *pointer-equality-operator*	pointer equality comparison
$\mid$ *pointer-relational-operator*	pointer relational comparison
$\mid$ `ptrdiff`	pointer subtraction
$\mid$ `intFromPtr`	cast of pointer value to integer value
$\mid$ `ptrFromInt`	cast of integer value to pointer value
$\mid$ `ptrValidForDeref`	dereferencing validity predicate

a ::= **memory actions**
$\mid$ `create` (pe_1,pe_2)
$\mid$ `alloc` (pe_1,pe_2)
$\mid$ `kill` (pe)
$\mid$ `store` (pe_1,pe_2,pe,*memory-order*)
$\mid$ `load` (pe_1,pe_2,*memory-order*)
$\mid$ `rmw` (pe_1,pe_2,pe_3,pe_4,*memory-order*$_1$,*memory-order*$_2$)

pa ::= **memory actions with polarity**	
$\mid$ a	positive, sequenced by both `let weak` and `let strong`
$\mid$ `neg` (a)	negative, only sequenced by `let strong`

pat ::=	
$\mid$ _	wildcard pattern
$\mid$ $ident$	identifier pattern
$\mid$ $ctor$(pat_1, .. ,pat_n)	constructor pattern

pe ::= **Core pure expressions**	
$\mid$ $ident$	Core identifier
$\mid$ <*impl-const*>	implementation-defined constant
$\mid$ $value$	value
$\mid$ `undef` (*ub-name*)	undefined behaviour
$\mid$ `error` ($string$,pe)	impl-defined static error
$\mid$ $ctor$(pe_1, .. ,pe_n)	constructor application
$\mid$ `case` pe `with` $\overline{\mid pat_i\texttt{=>}pe_i}^{\,i}$ `end`	pattern matching
$\mid$ `array_shift`(pe_1,$ctype$,pe_2)	pointer array shift
$\mid$ `member_shift`(pe,tag.$member$)	pointer struct/union member shift
$\mid$ `not` (pe)	boolean not
$\mid$ pe_1 $binop$ pe_2	binary operators
$\mid$ (`struct` tag){ $\overline{.member_i = pe_i}^{\,i}$ }	C struct expression
$\mid$ (`union` tag){$.member = pe$}	C union expression
$\mid$ $name$(pe_1, .. ,pe_n)	pure Core function call
$\mid$ `let` $pat = pe_1$ `in` pe_2	pure Core let
$\mid$ `if` pe `then` pe_1 `else` pe_2	pure Core if
$\mid$ `is_scalar`(pe)	
$\mid$ `is_integer`(pe)	
$\mid$ `is_signed`(pe)	
$\mid$ `is_unsigned`(pe)	

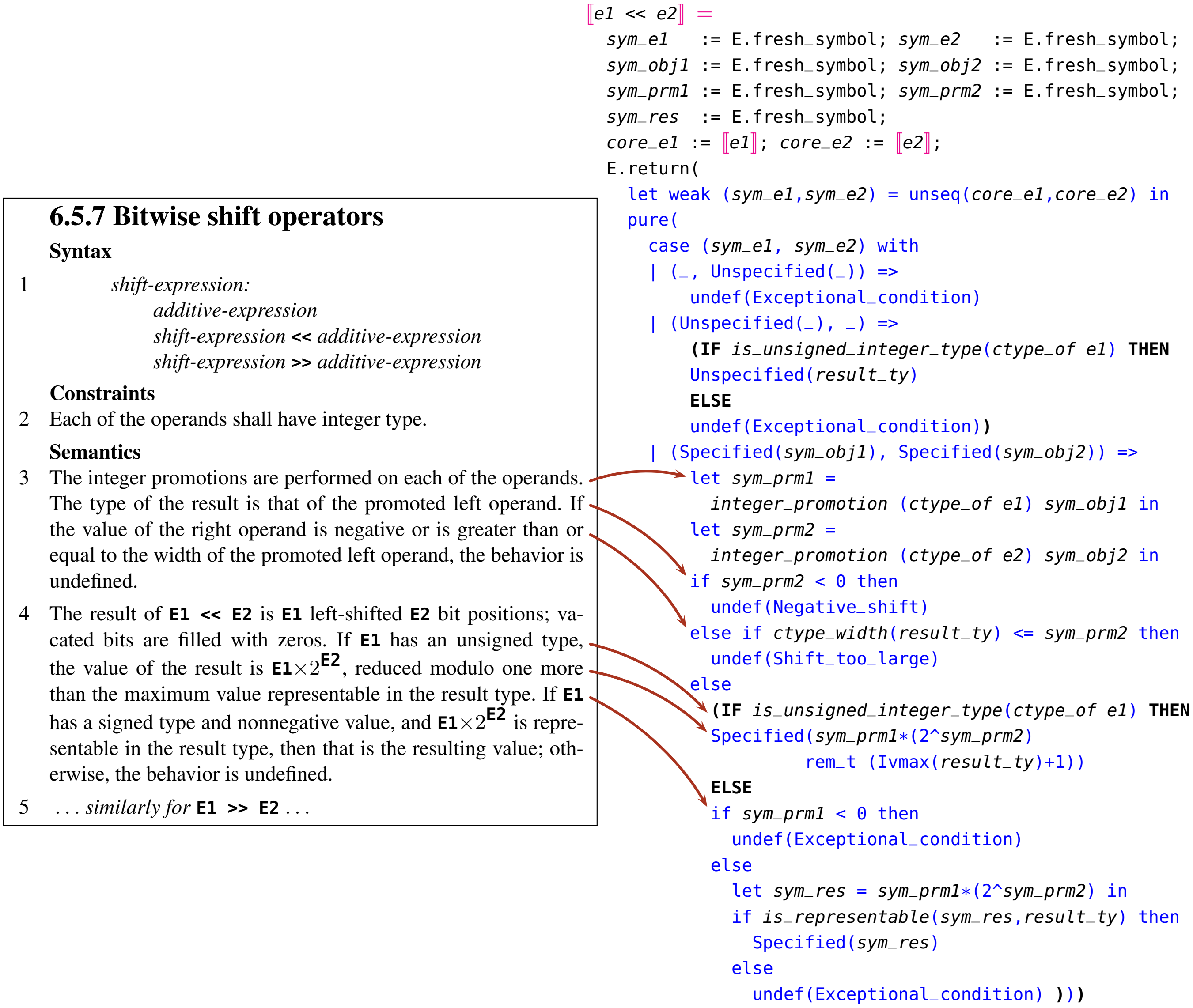
e ::= **Core expressions**	
$\mid$ `pure` (pe)	pure expression
$\mid$ `ptrop` ($ptrop$,pe_1, .. ,pe_n)	pointer op involving memory
$\mid$ pa	memory action
$\mid$ `case` pe `with` $\overline{\mid pat_i\texttt{=>}e_i}^{\,i}$ `end`	pattern matching
$\mid$ `let` $pat = pe$ `in` e	Core let
$\mid$ `if` pe `then` e_1 `else` e_2	Core if
$\mid$ `skip`	skip
$\mid$ `pcall` (pe,pe_1, .. ,pe_n)	Core procedure call
$\mid$ `return` (pe)	Core procedure return
$\mid$ `unseq` (e_1, .. ,e_n)	unsequenced expressions
$\mid$ `let weak` $pat = e_1$ `in` e_2	weak sequencing
$\mid$ `let strong` $pat = e_1$ `in` e_2	strong sequencing
$\mid$ `let atomic` (sym : oTy) = a_1 `in` pa_2	atomic sequencing
$\mid$ `indet` [n](e)	indeterminately sequenced expr
$\mid$ `bound` [n](e)	. . .and boundary
$\mid$ `nd` (e_1, .. ,e_n)	nondeterministic sequencing
$\mid$ `save` $label$($\overline{ident_i : ctype_i}^{\,i}$) `in` e	save label
$\mid$ `run` $label$($\overline{ident_i := pe_i}^{\,i}$)	run from label
$\mid$ `par` (e_1, .. ,e_n)	cppmem thread creation
$\mid$ `wait` (*thread-id*)	wait for thread termination

$definition$ ::= **Core definitions**	
$\mid$ `fun` $name$($\overline{ident_i : bTy_i}^{\,i}$) : bTy := pe	Core function definition
$\mid$ `proc` $name$($\overline{ident_i : bTy_i}^{\,i}$) : `eff` bTy := e	Core procedure definition

The result of elaborating a C program is a set of Core declarations together with the name of the startup (main) function; a set of struct and union type definitions; a set of names, core types, and allocation/initialisation expressions for C objects with static storage duration; the definitions of implementation-defined constants (some of which are Core functions); and a library of Core utility functions and procedures used by the elaboration.

Here tag, $member$, *memory-order*, *pointer-equality-operator*, and *pointer-relational-operator* are as in the C syntax, $ctype$ ranges over representations of C type expressions, $label$ ranges over C labels, and *ub-name* ranges over identifiers for C undefined behaviours. The $ident$ are Core identifiers, and $name$ ranges over C function names, Core function/procedure names, and implementation-defined constant names <*impl-const*>. The $binop$ and $ctor$ range over Core binary operations and value constructors (corresponding to the Core value productions). n and *thread-id* are natural numbers. Finally, $intval$, $floatval$, $ptrval$, and $memval$ are the representations of values from the memory layout model, containing provenance information as appropriate and symbolically recording how they are constructed (these are opaque as far as the rest of Core is concerned). The figure shows the concrete syntax for Core used by the tool; it is mechanically typeset from an Ott grammar that also generates the Lem types used in the model for the Core abstract syntax. We have elided Core type annotations in various places, C source location annotations, and constructors for values carrying constraints.

Figure 2. Core syntax



6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of **E1 << E2** is **E1** left-shifted **E2** bit positions; vacated bits are filled with zeros. If **E1** has an unsigned type, the value of the result is $\mathbf{E1}\times2^{\mathbf{E2}}$, reduced modulo one more than the maximum value representable in the result type. If **E1** has a signed type and nonnegative value, and $\mathbf{E1}\times2^{\mathbf{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 . . . *similarly for* **E1 >> E2** . . .

```
⟦e1 << e2⟧ =
 sym_e1   := E.fresh_symbol; sym_e2   := E.fresh_symbol;
 sym_obj1 := E.fresh_symbol; sym_obj2 := E.fresh_symbol;
 sym_prm1 := E.fresh_symbol; sym_prm2 := E.fresh_symbol;
 sym_res  := E.fresh_symbol;
 core_e1 := ⟦e1⟧; core_e2 := ⟦e2⟧;
 E.return(
  let weak (sym_e1,sym_e2) = unseq(core_e1,core_e2) in
  pure(
   case (sym_e1, sym_e2) with
   | (_, Unspecified(_)) =>
     undef(Exceptional_condition)
   | (Unspecified(_), _) =>
     (IF is_unsigned_integer_type(ctype_of e1) THEN
     Unspecified(result_ty)
     ELSE
     undef(Exceptional_condition))
   | (Specified(sym_obj1), Specified(sym_obj2)) =>
     let sym_prm1 =
      integer_promotion (ctype_of e1) sym_obj1 in
     let sym_prm2 =
      integer_promotion (ctype_of e2) sym_obj2 in
     if sym_prm2 < 0 then
      undef(Negative_shift)
     else if ctype_width(result_ty) <= sym_prm2 then
      undef(Shift_too_large)
     else
      (IF is_unsigned_integer_type(ctype_of e1) THEN
      Specified(sym_prm1*(2^sym_prm2)
          rem_t (Ivmax(result_ty)+1))
      ELSE
      if sym_prm1 < 0 then
       undef(Exceptional_condition)
      else
       let sym_res = sym_prm1*(2^sym_prm2) in
       if is_representable(sym_res,result_ty) then
        Specified(sym_res)
       else
        undef(Exceptional_condition) )))
```

The elaboration ⟦·⟧ is a Lem function that calculates the Core expression that a C expression or statement elaborates to. Here we show the definition for C left-shift expressions `e1 << e2`, described in Section 6.5.7 of the ISO C11 standard on the left. The elaboration, on the right, starts by constructing fresh Core identifiers and recursively calculating the elaborations of `e1` and `e2`, in a Lem monad for fresh identifiers. In the main body, the `IF ... THEN ... ELSE ...` are Lem conditionals, executed at elaboration time, while the lower-case blue keywords are parts of the calculated Core expression, constructors of the Lem types for the Core AST that are executed at runtime by the Core operational semantics. The former make use of two Lem functions: `ctype_of` and `is_unsigned_integer_type`. C expressions are effectful and so the calculated elaborations of the two operands (`core_e1` and `core_e2`) are impure Core expressions that need to be sequenced in some way. This is specified elsewhere in the standard, not in 6.5.7: Clause 6.5p1 says *"value computations of the operands of an operator are sequenced before the value computation of the result of the operator"*, modelled with the `let weak`; and Clause 6.5p2 states that *"side effects and value computations of subexpressions are unsequenced"*, captured by the `unseq`. The remainder of the elaboration only performs type conversions and arithmetic calculations and so is a pure Core expression. The `case` expresses our chosen de facto answers to **Q43** and **Q52**: unspecified values are considered daemonically for identification of possible undefined behaviours and are propagated through arithmetic; the rest captures the ISO left-shift text point-by-point, as shown by the arrows. Clause 6.5.7p2 is captured in our typechecker, not in the elaboration (`result_ty` is the appropriate promoted C type). The calculated Core contains some Core function calls to auxiliaries: `integer_promotion`, `ctype_width`, and `is_representable`. We elide details in the Lem development of the formation of these calls and the construction of Core type annotations.

Figure 3. Sample extract of C11 standard and the corresponding clause of the elaboration function

novel constructs to express the C evaluation order, a goto annotated with information about the C block boundary traversals involved, and nondeterministic choice and parallelism; all these are explained below (these last three are also considered to be effectful).

5.3 A Sample Excerpt of the Elaboration

In Fig. 3 we show a sample excerpt from the C11 standard, for left-shift expressions, and the corresponding clause of our elaboration function, mapping Typed Ail left-shift expressions into Core. It uses our constructs for weak sequencing (`let weak`), unsequencing (`unseq()`), and undefined behaviour (`undef()`). These are explained below, but one can already see that the elaboration and the standard text are close enough to relate one to the other — as one would hope, as this is one of the clearer parts of the standard. Making the whole of Cerberus accessible to a motivated but practical audience will require further work on presentation, but we believe this gives some grounds for optimism.

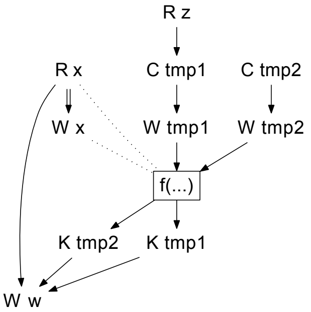
5.4 Undefined Behaviour

Undefined behaviour can arise dynamically in two ways: where a primitive C arithmetic operation has undefined behaviour for some argument values, and from memory accesses (unsafe memory accesses, unsequenced races, and data races). For the former, our elaboration simply introduces an explicit test into the generated Core code, as in the several uses of `undef()` in Fig. 3. If the Core operational semantics reaches one of these it terminates execution and reports which undefined behaviour has been violated (together with the C source location). This is analogous to the insertion of runtime checks for particular undefined behaviours during compilation, as done by many tools, except that (a) it is more closely tied to the standard, and (b) in Cerberus's exhaustive mode, it can detect undefined behaviours on any allowed execution path, not just those of a particular compilation. The latter are detected by the memory object or concurrency models, using calculated sequenced-before and happens-before relations over actions.

5.5 Arithmetic

The many C integer types (`char`, `short`, `int`, `int32_t`, etc.) cause frequent confusion and compiler errors [42], with their various finite ranges of values, signed and unsigned variants, different representation sizes and alignment constraints, and implicit conversions between integer types. For example, the C expression `-1 < (unsigned int)0`, perhaps surprisingly, can evaluate to `0` (false) (and typically does on x86-64). All this is captured in Cerberus by the elaboration, case-splitting over the inferred types as necessary, with Core computation simply over the mathematical integers.

5.6 Sequencing



The C standard is quite precise when it comes to the evaluation order of expressions, but it is a loose specification. For example, consider the C statement `w = x++ + f(z,2);`. Its sequencing semantics is shown on the right as a graph over the memory actions (reads R, writes W, object creations C, and object kills K); the solid arrows represent the *sequenced before* relation of the standard's concurrency model; the double arrow additionally links two actions into an atomic unit, and the dotted lines indicate *indeterminate sequencing*. Here (1) the evaluations of the operands of + are unsequenced with respect to each other; (2) the read of `x` and the body of `f()` are sequenced before the store to `w`; (3) the read and write of `x` are atomic, preventing other memory actions occurring between them; (4) for each argument of `f()`, first it is evaluated, then a temporary object is created and the value of the argument written to it; all that is before the body of `f()` but unsequenced with respect to other arguments; and (5) after the body of `f()` the temporary objects are killed before the return value is used in the write to `w`. Finally, (6) the body of `f()` is *indeterminately* sequenced with respect to everything with which it would otherwise be unsequenced, namely the read and write of `x`. This means that in any execution and in any occurrence of the statement, the body of `f()` is sequenced-before or sequenced-after (one way or the other) with respect to each of those two accesses; as the latter are atomic, it must be sequenced one way or the other with respect to both.

To capture this cleanly, we introduce several novel sequencing forms into the Core expression language.

$$\begin{array}{ll}
e ::= \ \ldots & \\
\mid\ \texttt{unseq}\,(e_1\texttt{,}\,..\,\texttt{,}e_n) & \text{unsequenced expressions} \\
\mid\ \texttt{let}\,\texttt{weak}\,pat = e_1\ \texttt{in}\ e_2 & \text{weak sequencing} \\
\mid\ \texttt{let}\,\texttt{strong}\,pat = e_1\ \texttt{in}\ e_2 & \text{strong sequencing} \\
\mid\ \texttt{let}\,\texttt{atomic}\,(sym : oTy) = a_1\ \texttt{in}\ pa_2 & \text{atomic sequencing} \\
\mid\ \texttt{indet}\,[n](e) & \text{indeterminate sequencing} \\
\mid\ \texttt{bound}\,[n](e) & \text{...and boundary} \\
\mid\ \texttt{nd}\,(e_1\texttt{,}\,..\,\texttt{,}e_n) & \text{nondeterministic seq.}
\end{array}$$

The first permits an arbitrary interleaving of the memory actions of each of the e_i (subject to their internal sequenced-before constraints) and reduces to a tuple of their values. The next two bind the result of e_1 to the identifiers in the pattern *pat* and to some extent sequence the memory actions of e_1 before those of e_2.

Consider the semantics of C assignments when combined with postfix increment and decrement. In specifying that evaluation of the right operand of an assignment occurs before the assigning store [52, §6.5.16#3], the standard restricts that ordering to the "value computation". Intuitively this is part of the expression evaluation impacting the final value, and in the postfix increment/decrement case, the "value computation" excludes the incrementing/decrementing store. To capture this in Core, we annotate memory actions with polarities; those that are not part of a value computation are negative. The strong sequencing operator makes all actions of e_1 sequenced-before all actions of e_2, while the weak sequencing operator only makes the positive actions of e_1 before the actions of e_2. Atomic sequencing is required only for the postfix increment and decrement operators: the `let atomic` makes the first memory action a_1 sequenced before the second pa_2 and prevents indeterminate sequencing putting other memory actions between them (shown with a double arrow on the left above). Finally, to calculate the possible sequenced-before orderings of any indeterminately sequenced function bodies in an expression, we have a construct to label subexpressions as indeterminately sequenced with respect to their context, and another to delimit the relevant part of that context (corresponding to the original C expression). A Core-to-Core transformation (expressed as a rewrite system using those operators) converts any expression into a nondeterministic choice between Core expressions that embodies all the possible choices; `indet` and `bound` do not appear in the result.

The point of all this expression-local nondeterminism is not to permit user-visible nondeterminism, but rather to permit optimisation. If two accesses to the same object are *unrelated* by the sequenced-before relation then they form an *unsequenced race* and the program has undefined behaviour (or, in other words, the compiler can assume that there are no two such accesses). Our semantics can detect all such unsequenced races.

5.7 Lifetime

Memory objects in C have lifetimes depending on their scope, storage class, and, for dynamically allocated objects, the calls to allocation and deallocation functions. A precise model of object lifetime is needed to detect illegal memory actions, including accessing objects outwith their lifetime and concurrency-related illegal actions such as data races.

Lifetime is made explicit in our elaboration into Core: Core has no notion of block, and memory objects are all created and killed using the explicit memory actions introduced by the elaboration. The Core `create` and `alloc` memory actions take an alignment constraint and either a `ctype` or allocation size respectively; they reduce to what corresponds to the C value of a pointer to the created object. The `load`, `store`, and `rmw` memory actions all take the `ctype` from the lvalue of the original C expression as their first arguments, together with values and addresses as appropriate. These are explicitly required to be sequenced, and then there are also *ptrop* pointer operations, which in some memory layout models need to access (or even mutate) the memory state, and so also must be sequenced.

5.8 Loops and Goto

Core provides recursive functions and a pure `if` construct. C statements, including `goto`, could be elaborated using these alone, with a CPS transformation of the C program, but this would lead to a serious loss of the source code structure and/or excessive code duplication, making it hard to relate the Core evaluation back to the original C program. We instead equip Core with labels and a primitive goto-like construct, giving semantics to those using continuations in the Core dynamic semantics.

This means that we can uniformly elaborate the various C control-flow constructs: the loops `for`, `while`, and `do`, the `break` and `continue` statements to prematurely exit a loop or skip an iteration, and the `switch` statement. All are elaborated using Core labels and the Core `save` and `run`, keeping the original code structure. For example, C of the form on the left below is elaborated into the Core on the right:

```
while (e) {       save l () in
  s1;               let strong id = ⟦e⟧ in
  break;            if id = 0 then
  s2;                 save b() in skip
};                  else
                      let strong _ = ⟦s1⟧ in
                      let strong _ = run b () in
                      let strong _ = ⟦s2⟧ in
                      run l()
```

Handling the interaction between the C `goto` and the lifetime of block-scoped variables in C requires the Core `goto` to do more. If a C `goto` is used to jump in the middle of a block, the local objects' lifetimes start when that jump is performed, as in the C `goto l; { int x = 0; l: S1; }`. Similarly, jumping out of a block ends the lifetime of the local objects. The target of any ISO C `goto` is statically determined (we do not support GCC computed `goto`s), so our elaboration can record sufficient information to handle this: it records the set of visible objects as it goes down through blocks, then it annotates the Core `save`s and `run`s with the Core symbolic names and `ctype`s associated with those objects. The dynamics of the Core `run` does a `create` for all the objects which are in scope at the target `save` but not at the `goto`, and a `kill` for all the converse.

5.9 Our Candidate De Facto Memory Object Model

Expressing the de facto memory object model that we sketched above requires enriched definitions for pointer, integer, and memory values. Pointer values and integer values all contain a provenance, either empty (for the NULL pointer and pure integer values), the original allocation ID of the object the value was derived from, or a wildcard (for pointers from IO). Most arithmetic involving one provenanced value and one pure value preserves the provenance, while subtraction of two values produces a pure integer (to use as an offset). Arithmetic involving two values with distinct provenance also produces a pure integer, to ensure the result cannot be used between them (this prevents the per-CPU-variable case without additional annotation). Pointer values also include either a symbolic base (essentially an allocation id) or a base cast from an integer value, together with a symbolic offset (to shift among struct members and arrays). Integer values can be concrete numbers, addresses of memory objects, symbolic arithmetic (for offsets), casts of pointer values to integers, pointer diffs, a byte of a memory value, bytewise composites, `sizeof`, `offsetof`, and `_Alignof`, and the maximum and minimum values of integer types. Memory values can be either unspecified, an integer value of a given integer type, a pointer, or an array, union, or struct of memory values.

We outline how this gives the desired behaviour for two of the questions from §2. For the basic provenance example of §2.1, it will construct fresh IDs at allocation time for the objects associated to `y` and `x`; the pointer values formed for `&x` and `&y` will have those provenances; and the pointer-plus-pure-integer arithmetic `&x+1` will preserve the `x` provenance. The `memcmp` will examine only the pointer representation bytes, not their provenances. Then one hits a tool design choice: one could either make nondeterministic choices for the concrete addresses of allocations (at allocation-time or more lazily) or treat them symbolically and accumulate constraints; both are useful. With the first, the `memcmp` can be concretely resolved to one of its three cases; with the second, one can make a nondeterministic choice between them and add the appropriate constraints (better for exhaustive calculation of all possible outcomes). In either semantics, following the "equals" branch, the semantics will check, at the `*p=11` access, whether the address of the pointer value (and width of the `int`) are within the footprint of the original allocation associated with the provenance of the pointer value. In this case it is not, so the semantics will flag an undefined behaviour and the tool will terminate.

For the pointer copying of §2.3, copying pointer values by copying their representation bytes (directly or indirectly) will work because those representation bytes (qua integer values) will carry the provenance of the original pointer value, and pure integer arithmetic will preserve that, as will the reconstruction of a pointer value that is read from multiple byte writes. Copying pointer values via control-flow choices will not because a control-flow choice on a pointer value, e.g. a C `if`, will not carry the pointer's provenance to values constructed in the "true" branch.

6. Validation

Cerberus is intended principally as a semantic definition that captures all the looseness of the C standard. We make it executable to explore the semantics of difficult test cases and enable comparison w.r.t. implementations, but that very looseness makes execution combinatorially challenging. It is not intended as a tool for production-sized C programs, but it does support small "real C" test cases. We validate this on the small tests from Ellison and Rosu [16]. Of their 561 Csmith tests [55], Cerberus currently gives the same result as GCC for 556; the other 5 time-out after 5min. The GCC Torture tests they mention are mostly not syntactically correct C11 programs, which we do not currently support. We also tried 400 larger Csmith tests, 40–600 lines long. Of these 22 did not terminate in GCC, presumably a bug in Csmith (the second Csmith bug we found). Cerberus terminates and agrees with GCC on 316, times out (after 30s) on 56 more, and fails on 6. Our de facto tests are much more demanding, and for these our candidate model, which is still work in progress, currently has the intended behaviour only for 9. Many of the failures are quite exotic, e.g. involving implementation-defined typing involving over-large constants, and questions about whether the result of `sizeof` is representable in types other than `size_t`, but there is still considerable work to do.

To demonstrate that Cerberus can be used as a basis for mechanised proof, we implemented a prototype translation validator, `tvc`, for the front-end of Clang. It supports only extremely simple single-function C programs that perform no I/O, take no arguments, and meet several additional restrictions. Given such a program, `tvc` produces a mechanised Coq proof that the behaviours of the IR produced by the compiler are a subset of those allowed by Cerberus (strictly, by the version current when `tvc` was produced). The former is defined by the Vellvm non-deterministic semantics of LLVM IR [56]; the latter the Cerberus elaboration and Lem's translation into Coq of the Core semantics. Although `tvc` is extremely limited, it shows that Cerberus can be used as the basis for mechanised proof about individual C programs, and it is notable in being a translation validator for the front end of a production compiler.

7. Related Work

Several groups have worked to formalise aspects of the ISO standards, including Gurevich and Higgens [17], Cook and Subramanian [13], Norrish [38, 39], Papaspyrou [40], Batty et al. [3], Ellison et al. [16, 18], and Krebbers et al. [25–29]. Krebbers [27, Ch.10] gives a useful survey. The main difference with our work is that we are principally concerned with the more complex world of the de facto standards, especially for the memory object model, not just the ISO standard; to the best of our knowledge no previous work has systematically investigated this.

Another important line of related work builds memory object models for particular C-like languages, including those for CompCert by Leroy et al. [30, 31], the proposals by Besson et al. [6, 7], the model used for seL4 verification by Tuch et al. [45], the model used for VCC by Cohen et al. [12], and the proposal by Kang et al. [23]. These are not trying to capture either de facto or ISO standards in general, and in most respects are aimed at rather better-behaved code than we see. CompCert started with a rather abstract block-ID/offset model [31]; the later proposals are relaxed to support more de facto idioms. Ellison and Rosu [16] also used a simple block model, while Hathhorn et al. [18] add some effective type semantics (though see §3), and Krebbers has adopted a very strong interpretation of effective types. We refer the reader to our design-space document [10] for more detailed comparison with related work, especially with respect to memory layout model issues.

None of the above addresses realistic general-purpose concurrency in C except that of Batty et al., who *only* study concurrency. The closest are the TSO model of CompCertTSO [46] and the KCC extension with TSO [19]. In contrast to C11 concurrency, supporting TSO can be done with little impact on the threadwise semantics.

Comparing with the KCC and Krebbers et al. work in more detail, there are also important differences in semantic style. Cerberus is expressed as a pure functional specification in Lem (extractable to OCaml for execution and to prover definitions for reasoning), and our factorisation via Core helps manage the complexity. Ellison et al. give a more monolithic semantics in the $\mathbb{K}$ rewriting framework, aiming primarily at executability. It is defined directly on C abstract syntax. Overall, the state of the dynamic semantics contains over 60 environments – viable for execution, but problematic for understanding or for proof. This semantics has been relatively well validated by testing against GCC behaviour. Krebbers et al. are focussed instead on metatheory, with type system, operational semantics, and program logics defined in Coq and mechanised theorems relating them. Similar to Cerberus, their dynamic semantics is factored via a core calculus, but one rather closer to C than our Core. They also describe an interpreter, extracted from Coq, used on "a small test suite for both defined and undefined behavior" [29] but apparently so far without more extensive validation.

In terms of coverage of C syntactic features, Cerberus lies between those two: it does not currently support bitfields, floats, `longjmp`, user-defined variadic functions, or multiple translation units, which Ellison et al. cover in some form [16, 18, Fig. 1] (though they are not capturing all the semantic looseness of C, e.g. treating float operations by executing them in a particular implementation).

Then there is a very extensive literature on static and dynamic analysis for C, and systems-oriented work on bug-finding tools (including tools such as Valgrind [36], Stack [48], and the Clang sanitisers). Exactly where each of these (and the models above) lie in our articulation of the design space is an interesting question for detailed future work, which the current paper enables.

8. Conclusion

The semantics of C has been a vexed question for much of the last 40 years, and the interactions between compiler optimisations, the corpus of systems code, analysis tools, and safety and security properties make it increasingly critical. Our multifaceted investigation into aspects of the de facto and ISO standards should help clarify the situation; it enables a range of future work on testing, semantics, analysis, verification, and standardisation.

Acknowledgements We thank all those who participated in our C surveys, without which this work would not have been possible. We thank John Regehr and Pascal Cuoq for running `tis-interpreter` tests and discussion of the results, Jean Pichon-Pharabod for other testing assistance, and Colin Rothwell, Jon Woodruff, Mike Roe, and Simon Moore for their work on the CHERI ISA and CHERI C. We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS Programme Grant), and a Gates Cambridge Scholarship (Nienhuis). This work is also part of the CTSRD projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. A non-final but recent version is available at `www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[2] ANSI. *Programming Languages – C: ANSI X3.159-1989*. 1989.

[3] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[4] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2), 2010.

[6] F. Besson, S. Blazy, and P. Wilke. A precise and abstract memory model for C using symbolic values. In *APLAS*, 2014.

[7] F. Besson, S. Blazy, and P. Wilke. A concrete memory model for CompCert. In *Proc. ITP*, 2015.

[8] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[9] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *Proc. 9th IEEE Intl. Working Conference on Source Code Analysis and Manipulation*, SCAM '09, 2009.

[10] D. Chisnall, J. Matthiesen, K. Memarian, K. Nienhuis, P., and R.N.M. Watson. C memory object and value semantics: the space of de facto and ISO standards, 2016. `http://www.cl.cam.ac.uk/~pes20/cerberus/`.

[11] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proc. ASPLOS*, 2015.

[12] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci. (SSV 2009)*, 254:85–103, October 2009.

[13] J. Cook and S. Subramanian. A formal semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, October, 1994.

[14] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[15] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *Proc. ASPLOS*, 2008.

[16] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proc. POPL*, 2012.

[17] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Proc. CSL '92*, 1993.

[18] C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. In *Proc. PLDI*, 2015.

[19] Charles McEwen Ellison III. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[20] Runtime Verification Inc. Rv-match v0.1. `https://runtimeverification.com/match/download/`, 2016. Downloaded 2016-03-11.

[21] Intel Plc. Introduction to Intel memory protection extensions, July 2013.

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

[23] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. In *Proc. PLDI*, 2015.

[24] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (ANSI C)*. Prentice Hall, 2nd edition, 1988.

[25] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *Proc. CPP, LNCS 8307*, 2013.

[26] R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *Proc. POPL*, 2014.

[27] R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015.

[28] R. Krebbers and F. Wiedijk. Separation logic for non-local control flow and block scope variables. In *FoSSaCS*, 2013.

[29] R. Krebbers and F. Wiedijk. A typed C11 semantics for interactive theorem proving. In *Proc. CPP*, 2015.

[30] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.

[31] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[32] K. Memarian, J. Matthiesen, K. Nienhuis, V. B. F. Gomes, J. Lingard, D. Chisnall, R. N. M. Watson, and P. Sewell. Cerberus, 2016. `www.cl.cam.ac.uk/~pes20/cerberus`, `www.repository.cam.ac.uk/handle/1810/255730`.

[33] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proc. ICFP*, 2014.

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

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

[36] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[37] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency, 2016. Draft available at `www.cl.cam.ac.uk/~pes20/cerberus`.

[38] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, U. Cambridge, Computer Laboratory, 1998.

[39] M. Norrish. Deterministic expressions in C. In *ESOP*, 1999.

[40] N. S Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, 1998.

[41] F. Pottier and Y. Régis-Gianas. Menhir. `gallium.inria.fr/~fpottier/menhir/`.

[42] J. Regehr. `blog.regehr.org/archives/721`, 2012.

[43] J. Regehr, April 2015. `blog.regehr.org/archives/1234`.

[44] TrustInSoft. `trust-in-soft.com/tis-interpreter`, 2015.

[45] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proc. POPL*, 2007.

[46] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[47] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? In *Proc. APSYS*, 2012.

[48] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proc. SOSP*, 2013.

[49] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, and R. Norton. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, November 2015.

[50] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy, SP*, 2015.

[51] WG14. Defect report summary for ISO/IEC 9899:1999.

[52] WG14. ISO/IEC 9899:2011.

[53] WG14. Defect report 260, September 2004. `www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm`.

[54] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA*, 2014.

[55] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. PLDI*, 2011.

[56] J. Zhao, S. Nagarakatte, M. M.K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. POPL*, 2012.