A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction

Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, Steve Zdancewic

A Two-Phase Infinite/Finite Low-Level Memory Model

Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction

CALVIN BECK, University of Pennsylvania, USA
IRENE YOON, Inria, France
HANXI CHEN, University of Pennsylvania, USA
YANNICK ZAKOWSKI, Inria, ENS de Lyon, CNRS, UCBL1, LIP, UMR 5668, France
STEVE ZDANCEWIC, University of Pennsylvania, USA

This paper provides a novel approach to reconciling complex low-level memory model features, such as pointer–integer casts, with desired refinements that are needed to justify the correctness of program transformations. The idea is to use a "two-phase" memory model, one with an unbounded memory and corresponding unbounded integer type, and one with a finite memory; the connection between the two levels is made explicit by a notion of refinement that handles out-of-memory behaviors. This approach allows for more optimizations to be performed and establishes a clear boundary between the idealized semantics of a program and the implementation of that program on finite hardware.

The two-phase memory model has been incorporated into an LLVM IR semantics, demonstrating its utility in practice in the context of a low-level language with features like `undef` and `bitcast`. This yields infinite and finite memory versions of the language semantics that are proven to be in refinement with respect to out-of-memory behaviors. Each semantics is accompanied by a verified executable reference interpreter. The semantics justify optimizations, such as dead-`alloca`-elimination, that were previously impossible or difficult to prove correct.

CCS Concepts: • **Theory of computation → Denotational semantics**; **Program verification**; **Program specifications**; • **Software and its engineering → Compilers**; **Semantics**.

Additional Key Words and Phrases: low-level memory model, integer–pointer casts, semantics, coq

ACM Reference Format:
Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (August 2024), 29 pages. https://doi.org/10.1145/3674652

1 Introduction

After 50 years the memory model for a programming language like C should be well understood! Unfortunately, memory models for low-level languages like C and LLVM IR are quite subtle and complex, especially when considered in the context of optimizations and program transformations [3–6, 13, 18–20, 22, 25, 26, 31, 32]. Why? These languages provide an abstract view of memory to justify a wide range of "high-level" optimizations—often pretending that available memory is unbounded and that allocations yield disjoint blocks, where a pointer to one allocated block can

Authors' Contact Information: Calvin Beck, University of Pennsylvania, Philadelphia, PA, USA, hobbes@seas.upenn.edu; Irene Yoon, Inria, Paris, France, euisun.yoon@inria.fr; Hanxi Chen, University of Pennsylvania, Philadelphia, PA, USA, hanxic@seas.upenn.edu; Yannick Zakowski, Inria, ENS de Lyon, CNRS, UCBL1, LIP, UMR 5668, France, yannick.zakowski@inria.fr; Steve Zdancewic, University of Pennsylvania, Philadelphia, PA, USA, stevez@cis.upenn.edu.

ACM 2475-1421/2024/8-ART263
https://doi.org/10.1145/3674652

never be used to access adjacent blocks from different allocations—but these languages also allow for low-level access to the memory, yielding a high degree of control and performance. Unfortunately, these two extremes are at odds, and it is difficult to ensure that the semantics of low-level memory operations preserve the invariants expected by the high-level optimizations.

This tension between low-level memory operations and high-level optimizations is evident in pointer arithmetic operations. Pointer arithmetic operations allow programmers to manipulate memory addresses as integer values, which exposes the underlying concrete memory layout. When the concrete memory layout is exposed, the behavior of a program can depend upon *where* values are allocated in memory, which can severely limit which optimizations can be performed. For instance, the assignment `a[i] = 2` *could* overwrite any other value in memory if `a[i]` is out of bounds of the array `a` (the underlying pointer arithmetic is hidden by the array index notation). That sounds reasonable when considering how this program would execute on a specific machine at a low level, but it is disastrous from the perspective of an optimizing compiler! Even if `a` is dead (i.e., never read from again), the compiler can't remove this store because it *might* alias with something that *is* live. To justify removing a (seemingly) dead store, the compiler would *also* have to prove that `i` is in bounds, but because `i` can be the result of an arbitrary computation this can be difficult, if not impossible. What is a compiler (or compiler implementor) to do?

Programming languages like C and LLVM use the notion of *"undefined behavior"* (or "UB") to justify the correctness of "high-level" optimizations without the need for complicated reasoning (like determining whether `a[i]` is in bounds). The compiler *assumes* that the program doesn't exhibit undefined behavior. For instance, by declaring that out-of-bounds accesses are UB, the compiler only needs to determine that `a` isn't read from again to justify deleting the dead `a[i] = 2` store—the compiler assumes that `a[i]` is in bounds, so we don't need to worry about it aliasing with anything except elements of `a`. Programs given to the compiler must not exhibit UB or the optimizations it performs won't be valid, potentially leading to unexpected results, but, in return, the compiler is able to perform much more aggressive optimizations.

While UB can be a very powerful tool, it can, unfortunately, be difficult to define the semantics of a programming language and its memory model such that situations that impact optimizations are classified as UB. This is particularly challenging in the context of more realistic memory models, such as those involving *finite* memory, in which seemingly pure operations may have visible side effects, like exhausting memory. For instance Lee, et al. [22] note that, in the case of finite memory, there is a side-channel that can be used to accurately guess the physical address of other blocks, making it difficult to rule out pointer aliasing brought about by casting arbitrary integers to pointers. Similarly (but perhaps counter-intuitively) in a finite memory model, it is, in general, *unsound* to remove a dead allocation operation: allocating less memory can turn a program that always runs out of space into one that makes more progress, thereby introducing more behaviors after optimization—to justify removing the allocation, the compiler would have to also prove that the allocation always succeeds! (This is the strategy taken in CompCertS [5].)

These situations may seem unimportant: Who cares if a program can determine the physical address of a block without directly observing it through a pointer–integer cast? Isn't the goal of removing an allocation to save memory, potentially allowing the program to make more progress? However, properly accounting for such semantics is essential for ensuring that the compiler makes consistent assumptions—inconsistency can lead to end-to-end miscompilation bugs and subtle erroneous interactions between optimization passes.[1] It is also vital in the context of formal

[1]There are numerous discussions about such semantics problems on the LLVM IR github issue tracker https://github.com/llvm/llvm-project/issues/. A few examples particularly germane to our focus are 54002, 55061, 52930, 33896, and, especially, 34577: *LLVM Memory Model needs more rigor to avoid undesired optimization results*, which has been open since 2017.

verification, which aims for optimizations to be "provably correct." For example, if the memory model does not rule out the possibility that a program can determine the physical address of some block, it can be impossible to justify an optimization that depends on memory locations being unaliased. Ultimately, an incoherent memory model leads to buggy software, and complicates formally verifying optimization passes.

Pointer–integer casts are a major sticking point for memory models [5, 18]. These casts expose the bare bones of the memory layout, which complicates alias analysis and can invalidate many optimizations, but these casts also bring a more subtle and sinister issue into play: cardinality. Most programming languages at this level of abstraction have integers with *finite* bitwidths; however, compilers and programmers pretend that there is no limit to the number of pointers a program can allocate, as doing so greatly simplifies reasoning (see the discussion in Lee, et al. [22]). This discrepancy between finitary integers and infinitary pointers means that one of the following design choices must be made: (1) a cast from a pointer to an integer can fail, (2) casting from a pointer to an integer and back does not necessarily yield the original pointer, and thus causes unexpected aliasing, or, (3) we must admit that memory is finite and make the pointer types finite as well. All of these options have implications for program transformations: (1) means that pointer–integer casts are effectful, instead of being (pure) no-ops, which means they can't easily be removed, (2) means that any code using a cast may result in pointers being truncated, which means it could cause aliasing, thereby invalidating many optimizations, and, (3) has many complicated consequences, including being unable to remove dead allocations (as mentioned above).

The big(int) idea. This paper proposes a new way to reconcile the desired features mentioned above. In particular, we present a semantics for low-level languages that provides an account of pointer–integer casts, while dealing with finite memory and justifying many desirable optimizations. The key insight is that, for such a language, there are really *two* memory models in play: one that assumes an unbounded amount of memory and the presence of an "unbounded integer type" (akin to `bigint`) for which there can be an injection from pointers to these unbounded integers, and a second that assumes a finite memory and in which all pointers and integer types have finite bitwidths. The advantage of this is that many optimizations, such as dead allocation elimination, are always valid in the infinite memory model without any additional reasoning (though some optimizations are valid in both). Optimizations should primarily be performed with respect to the infinite memory model, and then the program should be translated safely to the finite memory model for subsequent lowering to machine code. This *explicit* translation step that converts the program from the infinite model to the finite one is (almost) just the identity translation on syntax. Semantically, it may introduce new out-of-memory behaviors, but otherwise, the translated code retains a precisely specified connection to its infinitary behavior. The upshot is that we can reason about the impact of a compiler's optimizations in phases, but still have end-to-end guarantees about a program's behaviors that are sufficient to ensure correct compilation. In contrast, existing models for languages like LLVM IR don't separate these phases, which muddles the semantics and makes the assumptions unclear. While our focus is on languages that straddle the barrier between low-level memory accesses and high-level optimizations like LLVM IR and C, these ideas should be relevant to all higher-level languages that assume there is infinite memory and must ultimately be compiled to run on machines with finite memory.

Although this idea seems simple at first blush, working through all of the many details turned out to be highly nontrivial—for instance, in the infinite phase, one must address[2] what it means to store an "infinitely wide" pointer into a data structure that is serialized into a form stored in memory. To handle that, we observe that the infinite model's behaviors in such cases are always

[2]Pun intended!

(eventually) lowered to the finite model's, which gives us a degree of wiggle room: in the infinite memory model, reading such an infinitely wide pointer "atomically" from memory can be lossless, but—even in the infinite model—reading such a stored pointer byte-by-byte can truncate the pointer. So long as the finite model refines these behaviors, it will behave "as expected."

In summary, this paper makes the following contributions:

- We explain the design rationale of this two-phase infinite/finite memory model in the context of related state-of-the-art memory models for languages that support integer–pointer casts. See Section 2.
- We formalize the proposed memory model in the Coq theorem prover. The axiomatic description allows for many possible implementations, and we use Coq's module features to share the definitions common to the infinite and finite models. Each model is parameterized by a few basic abstractions (*symbolic bytes* and *addresses*) that should be relevant in any low-level programming language. The basic interface is given by just seven byte-oriented operations, which can be used to build methods for working with aggregate data such as vectors or structs. See Section 3.
- We define the relevant notions of simulation in both the infinite and finite models and prove that the translation from the infinite to the finite phase is a suitable refinement, introducing only new out-of-memory behaviors.
- To demonstrate the suitability of these ideas for modeling "real world" languages, we instantiate the memory models in the context of an existing formal semantics VIR, for LLVM IR, based on the work by Zakowski, et al. [39]. The VIR semantics aims to be a specification for a large, practically applicable subset of LLVM IR, but its prior memory models suffered from the deficiencies with respect to optimization correctness mentioned above. We choose this setting because VIR (and LLVM IR) supports a rich, C-like structured memory model, including integer–pointer casts; it also includes `undef` and `poison` values that interact non-trivially with the specification because they introduce nondeterminism and affect the notion of Undefined Behavior. Our instantiation handles a superset of the features supported by previous versions of VIR, giving us confidence that our memory model scales to realistic features sets. See Section 4.
- Besides the *specification* of the memory models, which is intended to be a logical (and hence nondeterministic) characterization of the set of allowed behaviors, we also define *executable implementations* for the VIR semantics (for both infinite and finite memories). We formally prove that these implementations refine the corresponding specifications. These executable semantics let us both test VIR against other LLVM IR implementations (specifically `clang`) and use Quickchick-style [12] randomized testing to probe the behaviors of our model.
- We further demonstrate the utility of this semantics for formal verification by proving the correctness of instances of dead-alloca elimination and dead `ptrtoint` cast elimination, which are representative of the reasoning needed to prove full-scale compiler optimizations such as register promotion and global-value numbering. We briefly describe these results in Section 5.

Although our proposal provides a piece of the puzzle for defining a "full" memory model for low-level languages like C or LLVM IR, we deliberately do not consider some features in this paper, leaving them to other (future) work. In particular, we omit concurrency altogether because there has already been much research on concurrent memory models, especially for relaxed-memory semantics, for such languages like C and the LLVM IR [1, 9, 17, 24, 34, 35]. Our treatment of infinite–finite refinement should be orthogonal to those proposals, but we expect it would take non-trivial engineering effort to combine them. We also elide and/or simplify some details of LLVM

Table 1. Comparison of various low-level memory models. Columns **PtoI** and **ItoP** describe pointer–to-int and int-to-pointer support, column **Finite** indicates support for finite memory, and **Ext. Mem.** describes whether the memory model requires extra memory.

Model	PtoI	ItoP	Finite	Ext. Mem.	Optimizations
Concrete	No-Op	No-Op	Yes	No	Bad
Logical Blocks	Unsupported	Unsupported	No	No	Good
Quasi-Concrete	Effectful	Yes	No, awkward	No	Good when no PtoI casts, cannot remove PtoI casts if concrete memory is finite
Twin-Allocation	No-Op	No-Op	Yes	Yes	Cannot remove dead allocations
CompCertS	No-Op	No-Op	Only Finite	Yes	Have to prove optimizations use less memory
Ours	No-Op	No-Op	2-Stages	No	Staged between infinite + finite compilation to allow more optimizations

IR that are not really relevant or that we expect to be straight forward to implement using this model; Section 4 describes the features we do consider.

Our formal semantics specification, the VIR implementations, and the claimed theorems are fully implemented and proved in the Coq interactive theorem prover. However, the full development is very large, relies heavily on Coq-specific details, and is a bit more general than what we need here. Thus, for the purposes of this paper, we have liberally simplified and streamlined the presentation, in-lining, renaming, and sharing some definitions when compared to the Coq code.[3]

2 Remembering Low-level Memory Models

To put our work in context, this section provides an overview of some basic memory models, focusing on which kinds of optimizations they allow, especially in the context of pointer arithmetic, pointer–integer casts, and finite memory. We'll start by reviewing a basic concrete memory model, compare it to a logical memory model (which does not support low level memory operations, but supports more optimizations), and then look at several memory models that bridge the gap between these two extremes, namely, the Quasi-Concrete model [18], the Twin-Allocation model [22], and the CompCertS finite memory model [5]. The summary of the comparison is given in Table 1.

2.1 Fully Concrete

One of the simplest memory models is a completely concrete one where memory is modeled by an array of bytes. Each allocation is assigned its own unique physical pointer which is just an integer index into the memory array. Modeling pointer–integer casts under this memory model is trivial as pointers really are just integer indices, making the casts no-ops. Furthermore, finite memory can easily be modified by simply restricting the size of the array.

This model of memory is perfectly reasonable, and is quite similar to how the memory in a physical computer actually

```
int main(int argc, char *argv[]) {
    char *a = malloc(4);
    char *b = malloc(4);
    *b = 1;
    char *c = a + f(0);
    *c = 2;
    // What can this print?
    printf("%d\n", *b);
    // optimized: printf("%d\n", 1)
    return 0; }
```

[3]The development can be found here https://github.com/vellvm/vellvm, and the artifact, with some additional documentation linking claims from the paper to the Coq development, can be found here https://zenodo.org/doi/10.5281/zenodo.12518800

works; unfortunately, these memory models are *too* concrete. The physical memory layout is not abstract at all, making it difficult to justify high level optimizations. For example, consider the program shown above.

It looks like `*b` is not modified after initialization, so it's sensible to use *store forwarding* to optimize away the extra load from `*b`, and replace the call to `printf` with `printf("%d\n", 1)` (which would enable further optimizations as `b` would now be dead). Unfortunately, the simple concrete memory model can't justify this optimization when `c == a + f(0) == b`, as this would mean that the write to `*c` overwrites the value stored in `*b`, and so it should print `2` instead of `1`. If we want to perform this optimization, we'll now have to know where `a` and `b` can be allocated in memory, and what the function call `f(0)` evaluates to. This is a lot of work for the compiler to justify a simple optimization, especially when the only reason it won't work is in the kind of degenerate case where you use `a` to generate a pointer to `b`, which should be out of bounds of `a`.

Undefined behavior. This is where undefined behavior (UB) comes into play. Language designers may decide that certain behaviors are "undefined," leaving the language semantics unspecified in such cases. The presence of UB justifies more aggressive compiler optimizations. For instance, in C the example program above is considered to have UB whenever `c` is a pointer outside of the region of memory allocated for `a`. The language implementation does not necessarily check for this UB; the compiler simply assumes that pointers constructed using pointer arithmetic stay in bounds of the original allocation, and thus `c` could never alias with `b`, because `b` was allocated with a different call to `malloc`. Compilers only need to preserve defined behavior, and so any case where UB occurs need not be considered when performing a program transformation. In the example above, we can perform store forwarding to have `printf("%d\n", 1)` instead—the only way that this program could print anything besides `1` is if `c` aliases with `b`, which would make the store to `*c` UB.

Unfortunately, this concrete memory model cannot justify such optimizations—the model is *too* well defined, giving a specific behavior to the program in the degenerate cases where out-of-bounds pointer arithmetic is used to overwrite arbitrary memory locations. We shouldn't be able to use a pointer from one allocation to derive an alias to a separate allocation. To address this we'd like to keep track of which pointers are allowed to access which regions of memory.

2.2 CompCert: Provenance

One way to solve the aliasing problem from the previous section is to give pointers *provenance*. The provenance of a pointer determines which block in memory that pointer is allowed to access. This provenance can be preserved throughout pointer arithmetic operations, so the pointer `c` should have the same provenance as `a`, and thus `c` should only be able to modify the block of memory associated with `a`, and cannot access the disjoint block of memory from the separate allocation `b`.

One example of a memory model that takes provenance into account is CompCert's [26, 27]. CompCert is a formally verified C compiler with an abstract-block-based memory model. Memory is no longer defined as a concrete array; instead memory is a map of blocks, and each allocation creates a block with a unique id `b` in the memory map. Pointers can then be represented by a tuple `(b, o)`, where `b` is the block id that serves as the provenance, and `o` is the offset into the block.

With this model, it's not possible for a pointer to be created that indexes into another block, as pointer arithmetic modifies only the offset and block ids never change. This is good news for the optimization in the example: `c` will never be able to alias with `b`. Unfortunately, it's not clear how we could handle casts between pointers and integers in this model because there is no longer a physical address for a block! Furthermore, because there is no physical memory layout, it is not clear how to implement finite memory in this case (one could limit the total size of the allocations, but without a physical layout of memory, it is difficult to take fragmentation into account).

2.3 Bringing Back Casting

There have been a couple of proposals for how to handle pointer–integer casting. The main two points of comparison are the quasi-concrete [18] and twin allocation [22] memory models.

2.3.1 Quasi-concrete Memory Model. The quasi-concrete memory model is an extension of CompCert's abstract-block style of memory models. The memory is split into two parts: logical, and concrete. The logical memory is the block/offset model described in Section 2.2, and, if no pointer–integer casts occur, the quasi-concrete memory is effectively identical to this model.

To support casts, the quasi-concrete memory model glues a concrete memory on top of the logical block model. This concrete layer represents the physical layout of the blocks in memory. Whenever a pointer is cast to an integer, a physical block is allocated in the concrete layer, representing where the logical block is *actually* allocated in physical memory. Delaying the allocation of a physical block until cast time can rule out situations where an address of a block might be guessed[4]. If the physical address of a block has never been observed through a integer cast, then a program should not be able to guess where that block is allocated (the block exists only in logical memory). Of course, while an actual program running on a real computer will allocate a physical address for every block immediately, the delayed allocation of physical blocks allows abstract pointers[5] to be completely isolated, such that physical addresses can never alias with them. Thus, even in the presence of complex casting between pointers and integers, more optimizations involving purely abstract pointers are justified, as are simpler heuristics for aliasing.

The downside of the delayed allocation of physical blocks is that pointer–integer casts have the side effect, within the semantics, of allocating a physical block in concrete memory. This means that we cannot erase any pointer to integer casts, even if they're dead, which in turn further restricts other optimization passes. For instance, an otherwise dead block of code or function call may need to remain in the program, because removing a cast will change the memory layout, impacting the behavior of the program. Removing the cast may mean the block is no longer accessible via integers cast to pointers, and may change where other blocks are allocated in concrete memory.

Furthermore, the story for finite memory is awkward in the quasi-concrete memory model. We can allocate as many logical blocks as we want, but, if there are a finite number of physical addresses, a cast from a pointer to an integer can cause an Out Of Memory (OOM) error. Ultimately, there are still all of the problems that we have with reasoning about finite programs, but they arise only in programs that perform pointer to integer casts.

2.3.2 Twin Allocation. Twin allocation [22] takes a different approach to handling pointer–integer casts, and does so while taking finite memory into account. Twin allocation gives every pointer a physical address immediately, but uses nondeterminism to rule out address guessing. Upon allocation, this memory model actually reserves *two* (or more) blocks instead of just one. One block is a "trap", and accessing it will raise UB; the other will be used as normal [6]. The model tracks two executions for the program nondeterministically, with the only difference between the executions being which of the two blocks is real and which is the trap. Then, if address guessing occurs, UB will be observed in one of the executions, as the guessed block will instead be a trap block in the alternate execution—and in that case, the program as a whole is considered to exhibit UB.

[4] An address is "guessed" if we construct a pointer to a block without deriving it directly from the allocation. This is mostly done via integer–pointer casts. For instance, casting an arbitrary integer to a pointer could give you an alias to any region of memory. Aliasing is problematic for optimizations, so we want to avoid it.

[5] Pointers whose physical address has never been observed via pointer–integer casts.

[6] Both blocks really do need to be allocated! If we just considered two executions, one where the block is allocated at `p1` and one where the block is allocated at `p2`, then it's plausible for something else to be allocated in the other slot for each of the executions, so you might not be able to swap `p1` for a trapped `p2`.

This model addresses some of the problems of the quasi-concrete memory model. Casts between pointers and integers aren't effectful and can thus be erased, as every block gets a physical address immediately. However, this model introduces some additional constraints on program transformations. Most importantly, allocations, even dead ones, cannot be removed! This issue is fundamental to the nature of finite memory models: when performing any allocation[7] in a finite model, the program may run out of memory, and, if it is removed, the program will behave differently—it might continue to execute instead of running out of memory. This situation isn't very satisfying, though, as programmers want the compiler to remove dead allocations! Section 3 will discuss our solution to this seemingly impossible conundrum.

Furthermore the twin-allocation model requires additional memory allocation to ensure that there's enough nondeterminism for address guessing to be detected. It should be possible for the extra allocation to be removed at run time, as that should also yield a valid execution of the program (the "trap" blocks can only cause UB to be raised sooner, or OOM), but it's awkward that we have to reason about programs with double (or more) of their actual memory usage. Section 6 of [31] makes the observation that it should be possible to instead reserve space for the largest allocation that the program can possibly make, instead of duplicating every allocation, which makes a slightly different reasoning trade-off. Using this strategy, one would have to prove that a program never performs an allocation larger than this pre-allocated trap block in order to guarantee that addresses are not guessed, which is an additional burden on the compiler, or on the programmer if such large allocations would be considered UB instead.

2.4 CompCertS: A Finite Symbolic Memory Model

CompCertS [5] extends the classic CompCert memory model with symbolic values (as in [3] and [4]), and allows for pointers to be treated as integers — our memory model takes a very similar approach with respect to the abstract bytes stored in memory as discussed in Section 4.2.2.

For our purposes, the most relevant aspect of CompCertS is how it handles finite memory. CompCertS makes the assumption that programs do not run out of memory, and any program transformation that CompCertS performs must be shown to either preserve, or decrease the amount of memory allocated by the program. These are perfectly reasonable design decisions, but this means that (1) to ensure correct compilation of a program, that program must be proven to not run out of memory, (2) any program transformations must be shown to not use additional memory, and (3) the finite memory address guessing side-channel discussed in Section 2.3.2 is present.

The constraints introduced by (2) can be mitigated somewhat by "pre-allocating" some unused memory that can be utilized by future program transformations, and it should also be possible to reclaim memory that is no longer needed. Program transformations that decrease memory usage should always be applicable (assuming the source program does not run out of memory), but transformations that increase memory usage may only apply conditionally. This approach to handling finite memory is honest and yields strong guarantees about the memory usage of the target program, but the restrictions on which optimizations can be performed are not ideal—ideally, we want to let our compiler hand-wave reasoning about memory altogether (in a semantically consistent way).

3 A Two-Phase Memory Model

To address the limitations of the memory models described in Section 2, our proposal is to use two phases of compilation to get the best of all worlds: an infinite memory model where high-level

[7] Assuming the bound on memory size is not known in advance, which is the usual assumption. If the size *is* known, then one could prove that some allocations will always succeed, but that poses other complications.

$$\sigma \in \mathit{Conf} \triangleq \begin{cases} \mathit{mem} & : \mathit{Memory}, \\ \mathit{heap} & : \mathit{Heap}, \\ \mathit{stack} & : \mathit{FrameStack}, \\ \mathit{used} & : \mathcal{P}(\mathit{Prov}) \end{cases}$$

$$\begin{aligned} m \in \mathit{Memory} &\triangleq \mathit{Addr} \hookrightarrow (\mathit{SByte} \times \mathit{Prov}) \\ h \in \mathit{Heap} &\triangleq \mathit{Addr} \hookrightarrow \mathcal{P}(\mathit{Ptr}) \\ p \in \mathit{Ptr} &\triangleq \{a : \mathit{Addr};\ \mathit{pr} : \mathit{Prov}\} \\ f \in \mathit{Frame} &\triangleq \mathcal{P}(\mathit{Ptr}) \\ \mathit{fs} \in \mathit{FrameStack} &\triangleq \texttt{list}\ \mathit{Frame} \\ \mathit{pr} \in \mathit{Prov} &\triangleq \mathit{option}(\mathbb{N}) \end{aligned}$$

Fig. 1. Datatype of memory configurations, where *Addr* and *SByte* are abstract parameters

abstract optimizations can be performed with ease, and a finite memory model that more closely represents the finite architecture of the compilation target. In our infinite model, both allocations and casts between pointers and integers can be removed (if they're dead) or added at will, so the presence of these operations doesn't block optimizations. Nearly all optimizations should be done under the infinite semantics, as optimizations that are valid under the finite model are also valid under the infinite model. Once optimizations are performed, there is an explicit translation to the finite model. That compilation step preserves the semantics of the original infinite program, but potentially introduces points where the program can halt early because it ran out of memory.

At a high level, the design of our two-phase memory model resembles the concrete memory models from Section 2.1 and Section 2.4. The only real difference between our infinite and finite models is the type of the pointers and the `iptr` type that we introduce in Section 4.2 in order to handle pointer / integer casts appropriately. The infinite model uses Coq's big-integer $\mathbb{Z}$ type for physical addresses, and the finite versions use an implementation of 64-bit integers (limiting the size of memory to the 64-bit address space). The `iptr` type matches the type of the physical addresses in the respective memory model.

The semantics for our memory model is nondeterministic, allowing us to accurately model the behavior of the program under the different memory layouts that arise from nondeterministic allocations. This nondeterminism can also be used to prevent address guessing in the infinite memory model, as there will always be an execution where a guessed block could be allocated somewhere else instead (akin to swapping blocks in the twin-allocation model, except no pre-allocation of these "trap blocks" is necessary because infinite memory means we always have unallocated space to swap blocks to).

3.1 Notations

We write $m : A \hookrightarrow B$ when m is a partial map from A to B. We write $m[a] = b$ to assert that a belongs to the domain of m and maps to b, $m\{a := b\}$ for updating a in m with value b, possibly extending the domain of m in the process, and $m \setminus a$ to remove a from the domain of m. When r is an element of a record type, we write $r.f$ for the content of its field f. We use the notation $\mathcal{P}(A)$ for the set of all subsets of elements of A. Given a list, l, we write $|l|$ for its length, $l[i]$ to access its i-th element, assuming it is within bounds, and coerce it into a set implicitly when needed. Finally, we conflate equality and extensional equality over finite maps.

3.2 Memory Configurations

Figure 1 describes the datatype *Conf* of memory configurations. It is parameterized by two types: *Addr*, the representation of concrete addresses, and *SByte*, the representation of (symbolic) bytes, in memory. The former is straightforward: addresses are represented as unbounded integers at the infinite level, and bounded integers at the finite level—we assume an operation $+ : \mathit{Addr} \to \mathbb{N} \to \mathit{OOM} + \mathit{Addr}$, which performs arithmetic on addresses returning *OOM* in the case where an overflow occurs in the finite model. We will leave the representation of *symbolic* bytes abstract,

as their implementation is language dependent, but we will give a full description of them in our LLVM semantics in Section 4.2.2; we invite the reader to think of them as concrete bytes until then.

A configuration σ has four fields. The memory itself, $\sigma.mem$, is a finite map from addresses to bytes with an associated provenance. The provenance is an optional natural number, where the `None` constructor is used as a wildcard during integer–pointer casts. We introduce notations to access the memory via pointers, i.e., addresses tagged with provenance information. We write $m[p] \doteq b$ for the partial allowed *SByte* lookup operation: it asserts both that $p.a$ is in the domain of m, *and* performs a provenance check by ensuring that it maps to the pair $(b, p.pr)$. We simply write $p \dot{\in} m$ as a shortcut to $\exists b, m[p] \doteq b$ to state that a pointer is accessible in memory. Conversely, $p \dot{\notin} m$ states that a pointer cannot be accessed in memory, either because $p.a$ is not in the domain of m, or because the associated provenance is different from $p.pr$. Finally, we write $m_1 \equiv_{\setminus \overrightarrow{p}} m_2$ to express that memories m_1 and m_2 agree on content and provenance at all addresses except those in the list of pointers $\overrightarrow{p}$:

$$m_1 \equiv_{\setminus \overrightarrow{p}} m_2 \triangleq \forall p', b, (\forall p \in \overrightarrow{p},\ p'.a \neq p.a) \rightarrow (m_1[p'] \doteq b \leftrightarrow m_2[p'] \doteq b)$$

The heap $\sigma.heap$ tracks information about heap allocation units. Via *malloc*, a contiguous region of memory can be allocated in *blocks* of sequentially consecutive pointers: $p_1, \ldots, p_n$. Each p_i : *Ptr* in the block is associated with its *root* address $p_1.a$, which is the address returned by the allocation operation. Freeing the root deallocates the whole block. (Freeing a non-root address will be undefined behavior.) The stack, $\sigma.stack$, keeps track of the call stack by maintaining a stack of *frames*, where each frame consists of a list of pointers. Fresh addresses, allocated via `alloca`, are added to the top frame of the stack, referred to as $\sigma.stack.top$. Finally, a configuration keeps track of a set of used provenances, $\sigma.used$ in order to ensure that fresh provenances can be assigned to new allocations.

3.3 The Specification Monad

Memory models for compiler IRs are naturally nondeterministic semantic objects: they must describe *all* legal implementations architectures may commit to, and allocations, in particular, are left unconstrained, leading to nondeterminism. We provide a specification for each operation via a *specification monad*:

$$MemSpec(X) \triangleq Conf \rightarrow \mathcal{P}(Result(Conf \times X)) \text{ where } Result(A) \triangleq UB + OOM + FAIL + ok(A)$$

$MemSpec(X)$ is stateful and nondeterministic, relating initial configurations to a set of possible configurations that could result from executing a memory operation. These sets are specified in a propositional way; we describe them either via inference rules, or by composing them via the usual `ret` and `bind` monadic operations. Finally, the result type, $Result()$, allows us to characterize the four possible acceptable behaviors of an operation. We write $(c\ \sigma) \ni beh$ to state that beh is a valid behavior of a memory specification c at initial configuration σ.

An operation may simply succeed, yielding $ok(\sigma, x)$, returning a new configuration σ and a resulting value x. It may also raise one of three exceptional behaviors. The first is *Undefined Behavior*, *UB*, which arises from run-time situations that invalidate assumptions the compiler makes to justify optimizations; semantically, these are modeled as computations that can be refined into anything. The second is an out of memory exception, *OOM*. This behavior captures all the situations in which the computation may preemptively halt as a consequence of the representation of addresses; semantically it is modeled as a behavior that refines anything. Lastly, operations may *FAIL*, representing cases in our semantics that we intend to be statically checked and ruled out. This exception also corresponds to language features that have not been implemented in our model. In

$$\begin{array}{rcl} \texttt{read}_b\ (p : Ptr) & : & MemSpec(SByte) \\ \texttt{write}_b\ (p : Ptr)\ (b : SByte) & : & MemSpec(unit) \\ \texttt{pushf} & : & MemSpec(unit) \\ \texttt{popf} & : & MemSpec(unit) \end{array} \qquad \begin{array}{rcl} \texttt{alloca}\ (\overrightarrow{b} : \texttt{list}\ SByte) & : & MemSpec(Ptr) \\ \texttt{malloc}\ (\overrightarrow{b} : \texttt{list}\ SByte) & : & MemSpec(Ptr) \\ \texttt{free}\ (p : Ptr) & : & MemSpec(unit) \end{array}$$

Fig. 2. Memory model: low level operations

$$\frac{p \notin \sigma.mem}{(\texttt{read}_b\ p\ \sigma) \ni UB} \qquad \frac{\sigma.mem[p] \doteq b}{(\texttt{read}_b\ p\ \sigma) \ni ok(\sigma, b)} \qquad \frac{p \notin \sigma.mem}{(\texttt{write}_b\ p\ b\ \sigma) \ni UB}$$

$$\frac{\begin{array}{c}\sigma_1.heap = \sigma_2.heap \quad \sigma_1.stack = \sigma_2.stack \quad \sigma_1.used = \sigma_2.used \\ p \in \sigma_1.mem \quad \sigma_2.mem[p] \doteq b \qquad \sigma_1.mem \equiv_{\setminus\{p\}} \sigma_2.mem\end{array}}{(\texttt{write}_b\ p\ b\ \sigma_1) \ni ok(\sigma_2, \texttt{tt})}$$

Fig. 3. Memory model: byte-level read and write operations

$$\frac{p \notin \sigma.used}{(\texttt{fresh}\ \sigma) \ni ok(\sigma\{used := \{p\} \cup \sigma.used\}, p)} \qquad \frac{\forall i,\ 0 \le i < n, \sigma.mem[a+i] = \texttt{None}}{(\texttt{find\_bk}\ n\ \sigma) \ni ok(\sigma, a)}$$

$$\frac{\begin{array}{c}\sigma_1.heap = \sigma_2.heap \quad \sigma_1.used = \sigma_2.used \quad \sigma_2.stack = \sigma_1.stack.top \uplus \overrightarrow{p} :: \sigma_1.stack.tl \\ |\overrightarrow{p}| = |\overrightarrow{b}| \quad \forall i, 0 \le i < |\overrightarrow{p}|,\ \sigma_2.mem[\overrightarrow{p}[i]] \doteq \overrightarrow{b}[i] \qquad \sigma_1.mem \equiv_{\setminus\overrightarrow{p}} \sigma_2.mem\end{array}}{(\texttt{alloca\_post}\ bs\ ptrs\ m_1\ ) \ni ok(m_2, \texttt{tt})}$$

$$\frac{\begin{array}{c}\sigma_1.stack = \sigma_2.stack \quad \sigma_1.used = \sigma_2.used \quad \sigma_2.heap = \sigma_1.heap\{\overrightarrow{p}[0].a := \overrightarrow{p}\} \\ |\overrightarrow{p}| = |\overrightarrow{b}| \quad \forall 0 \le i < |\overrightarrow{p}|,\ \sigma_2.mem[\overrightarrow{p}[i]] \doteq \overrightarrow{b}[i] \qquad \sigma_1.mem \equiv_{\setminus\overrightarrow{p}} \sigma_2.mem\end{array}}{(\texttt{malloc\_post}\ \overrightarrow{b}\ \overrightarrow{p}\ \sigma_1\ ) \ni ok(\sigma_2, \texttt{tt})}$$

$$\frac{\begin{array}{c}\sigma_1.stack = \sigma_2.stack \quad \sigma_1.used = \sigma_2.used \quad \sigma_1.heap[p.a] = \texttt{Some}\ \overrightarrow{p} \quad \sigma_2.heap = \sigma_1.heap \setminus \{p.a\} \\ \forall p' \in \overrightarrow{p},\ (p' \dot\in \sigma_1.mem \wedge\ \sigma_2.mem[p'.a] = None) \qquad \sigma_1.mem \equiv_{\setminus\overrightarrow{p}} \sigma_2.mem\end{array}}{(\texttt{free}\ p\ \sigma_1) \ni ok(\sigma_2, \texttt{tt})}$$

Fig. 4. Memory model: memory management primitives

the remainder of the paper, we therefore elide this case by providing partial specifications instead. A key distinction between failure and *UB* is that failure is not "time-traveling" in our semantics.

The monadic specification is particularly useful for maintaining very similar structures between the memory model and the executable interpreter (See Section 6), which simplifies maintenance and the proof that the executable implementations of the memory model respect the specifications.

3.4 Low Level Memory Operations

Signatures of the low level primitives interacting with the memory model are described in Figure 2. The operations are reads and writes of single bytes, allocations of blocks (a contiguous sequence of bytes) on the stack and heap, operations for freeing heap allocated blocks, pushing stack frames, and popping the most recent stack frame in order to free stack allocated blocks.

$$\frac{\sigma_1.heap = \sigma_2.heap \quad \sigma_1.used = \sigma_2.used \quad \sigma_1.mem = \sigma_2.mem \quad \sigma_2.stack = \emptyset :: \sigma_1.stack}{(\texttt{pushf}\ \sigma_1) \ni ok(\sigma_2, \texttt{tt})}$$

$$\frac{\begin{array}{c}\sigma_1.heap = \sigma_2.heap \quad \sigma_1.used = \sigma_2.used \quad \sigma_1.stack = \overrightarrow{p} :: \sigma_2.stack \\ \forall p \in \overrightarrow{p},\ (p \mathrel{\dot\in} \sigma_1.mem \wedge\ \sigma_2.mem[p.a] = None) \quad \sigma_1.mem \equiv_{\setminus \overrightarrow{p}} \sigma_2.mem\end{array}}{(\texttt{popf}\ \sigma_1) \ni ok(\sigma_2, \texttt{tt})}$$

Fig. 5. Memory model: frame stack management

Out of memory behavior. Perhaps surprisingly we *always* allow operations to halt preemptively with an out-of-memory behavior. One of our goals was for the specifications to encompass a large number of possible implementations for memory, and we've seen instances of memory models that might "run out of memory" in counter-intuitive situations (for instance a quasi-concrete memory model with a finite concrete memory may run out of memory when a concrete block is allocated for a pointer–integer cast), as such we've been very lenient with allowing out-of-memory behaviors throughout or semantics. One could tighten the specifications if desired, though when comparing the infinite and finite memory models in Section 3.5 our refinement relations allow for out-of-memory anywhere in the finite memory model anyway. Since out-of-memory is omnipresent, we work under the convention that all specifications $c : MemSpec(X)$ may run out of memory, i.e., $(c\ m) \ni OOM$ is satisfied for any initial memory m. This makes OOM a kind of refinement dual to UB. While UB can always be refined into any computation, any computation may be refined by OOM, as we purposefully do not want to reason about programs that run out of memory.

Reading and writing bytes (Figure 3). The operation $\texttt{read}_b\ p\ \sigma$ specifies the possible behaviors when dereferencing a pointer in memory. Dereferencing boils down to looking up the memory, with the additional provenance check introduced in Section 3.2. If the lookup fails, or is illegal, UB may be raised. Writing a byte b to memory $\sigma_1.mem$ at a pointer p may trigger UB in similar cases to reading. A successful write must furthermore specify the resulting configuration σ_2—the statement is made slightly more complex to account for provenance, but hides no surprise. Only the memory component of the configuration is modified. A pointer p accessible in $\sigma_1.mem$ will be remapped to the byte b in $\sigma_2.mem$. Finally, all addresses distinct from $p.a$ are unchanged in memory.

Memory management (Figure 4). Specifying allocation is a more involved and less atomic task, so we leverage the specification monad to describe it in a more programmatic style. Utility specifications help generate a fresh provenance pr (`fresh`), and retrieve an available contiguous sequence of n addresses in memory (`find_bk` n). [8] Note that `find_bk` uses $a + i$ to compute contiguous addresses, should this overflow in the finite model the allowed behavior will only be OOM.

Stack and heap allocations of a list of bytes are specified very similarly, retrieving a fresh provenance, an available range of addresses, constraining the resulting memory, and finally returning the first allocated address:

```
alloca (b⃗ : list SByte) : MemSpec(Ptr) :=
  pr ← fresh;;
  a ← find_bk (|b⃗|);;
  alloca_post b⃗ [(a, pr), ..., (a + |b⃗| − 1, pr)] ;;
  ret (a, pr)
```

```
malloc (b⃗ : list SByte) : MemSpec(Ptr) :=
  pr ← fresh;;
  a ← find_bk (|b⃗|);;
  malloc_post b⃗ [(a, pr), ..., (a + |b⃗| − 1, pr)] ;;
  ret (a, pr)
```

[8] In line with LLVM, allocating no bytes (i.e. an empty list) may return *any* address. Its fresh provenance, which no data is equipped with, will, however, ensure we cannot do anything with this address without triggering UB.

$$\sigma^{inf} \gtrsim \sigma^{fin} \quad := \quad \sigma^{inf} = \lceil \sigma^{fin} \rceil \qquad \text{Refinement}$$

$$\lceil \sigma^{fin} \rceil \quad := \quad \begin{cases} mem & = & \{z \mapsto (\lceil b \rceil, p) \mid \sigma^{fin}.mem[z] = (b,p)\} \\ heap & = & \{z \mapsto \mathtt{map}\ \lceil - \rceil\ \overline{blk} \mid \sigma^{fin}.heap[z] = \overline{blk}\} \\ stack & = & \mathtt{map}\ (\lambda \overline{blk} \cdot \mathtt{map}\ \lceil - \rceil \overline{blk})\ \sigma^{fin}.stack \\ used & = & \sigma^{fin}.used \end{cases} \qquad \text{Configuration Lifting}$$

$$\begin{array}{rcl} \lceil p \rceil & := & (\mathtt{int\_to\_}\mathbb{Z}(p.a), p.pr) \\ \lceil b \rceil & := & \mathtt{lift\_sbyte}(b) \quad (\textit{lifting symbolic bytes, language specific}) \end{array}$$

Fig. 6. Infinite-to-finite refinement defined by lifting of finite memory configurations into infinite memory configurations. Here, `lift_sbyte` is a language-specific lifting of finite $SByte^{fin}$ into $SByte^{inf}$. We omit implicit `int_to_`$\mathbb{Z}()$ casts when the integer is known to be within finite bounds.

These specifications only differ in how the configurations are constrained as depicted on Figure 4. For stack allocation, `alloca_post` $\overrightarrow{b}$ $\overrightarrow{p}$ ensures that (1) the new pointers are added to the current stack frame, (2) the addresses are written with the corresponding bytes, all sharing the provenance pr, and (3) nothing else in memory is altered. For heap allocation, `malloc_post` $\overrightarrow{b}$ $\overrightarrow{p}$ enforces (2) and (3) similarly, but, instead of manipulating the stack, it ensures that the returned address is a root in the heap associated with the set of newly allocated pointers.

Given a pointer p, `free` p σ_1 ensures that $p.a$ is a valid root in $\sigma_1.heap$, associated with a set of pointers $\overrightarrow{p}$ accessible in $\sigma_1.mem$. Under this assumption, it simply reclaims the pointers and severs $p.a$ from the heap. Though not shown in the Figure, UB occurs in the complementary cases—when the pointer is not a root in the heap or if the block was not actually allocated in memory.

Stack management (Fig 5). Pushing a new frame, `pushf`, is trivial, simply adding the emptyset on top of the stack. The specification of `popf` is very close to `free`: we ensure all pointers in the top frame are accessible in the original memory, reclaim them, and pop the stack.

3.5 Relating the Infinite and Finite Memory Models

We now consider two instances of the memory model, based on two representations of addresses: an *infinite* memory model with unbounded integers, and a *finite* one with 64-bit integers. We will distinguish between these models with *inf* and *fin* superscripts for the infinite and finite memory models respectively.

Our overarching goal is to be able to reason about programs in the infinite memory model, performing program transformations under the infinite semantics. We will then convert these infinite memory programs to finite memory programs that will eventually be compiled to native assembly code for a concrete architecture. In order to ensure that this process is sound, we must relate the behavior of memory operations under the infinite model with the behavior of the same operations under the finite model.

The rough idea is to consider the execution of programs under the infinite model when their allocations happen to fit within the finite memory model's range of memory addresses—executions that don't fit will exhibit *OOM* behavior. We then ensure that operations on these finite memory slices agree between the infinite and finite memory models. Essentially, the finite memory model should be able to simulate the infinite memory model, as long as all the addresses stay in bounds!

3.5.1 Relating Configurations. We start by relating configurations at both levels. As described in Figure 6, a finite $\sigma^{fin} : Conf^{fin}$ is a refinement of an infinite σ^{inf}, written $\sigma^{inf} \gtrsim \sigma^{fin}$, when

σ^{inf} coincides with the lifting $\lceil \sigma^{fin} \rceil$. Lifting a $Conf^{fin}$ is fairly straightforward, both domains of configurations having similar concrete representations. The lifting therefore simply maps over the structure the trivial injection of finite addresses into $\mathbb{Z}$, as well as the lifting of symbolic bytes.[9]

3.5.2 Relating Operations. We can now capture the intuitive expected behavior of the memory operations: if the memory configuration can fit in the finite representation, then the same behavior can be observed. In practice, we prove a refinement lemma for each low-level operation. For instance, in the case of reads:

Lemma 3.1 (`read_byte_spec` refinement).

If
- $\sigma^{inf} \gtrsim \sigma^{fin}$, *and*
- $ptr^{inf} = \lceil ptr^{fin} \rceil$, *and*
- $(\texttt{read}_b\ \sigma^{inf}\ ptr^{inf}) \ni ok(\sigma^{inf}, b^{inf})$

then $\exists\ b^{fin}$ *such that*
- $b^{inf} = \lceil b^{fin} \rceil$, *and*
- $(\texttt{read}_b\ \sigma^{fin}\ ptr^{fin}) \ni ok(\sigma^{fin}, b^{fin})$.

The lemmas for the other operations are similar in shape, so we omit them for conciseness.

4 Integrating the Memory Model into an LLVM like Language

To demonstrate the usefulness and expressiveness of our memory model described above, we incorporate it into a modified version of the VIR [39] formal semantics. VIR is a specification for a large, practically applicable subset of (sequential) LLVM IR. It supports a rich, C-like structured memory model, including integer–pointer casts; it also includes `undef` and `poison` values that interact non-trivially with the memory model specification because they affect the notion of UB. Incorporating the two-phase memory model into VIR increases the accuracy of the semantics, and allows for more optimizations to be formally justified. In addition to integrating our memory model with VIR, we have made further changes to the VIR semantics. The structure of the interpretation layers, as discussed in the next section, has been changed to incorporate the non-determinism in the memory model, we have fixed the behaviour of non-deterministic values with respect to various operations, and changes have been made to how UB is handled in order to bring the semantics further in line with LLVM proper. Furthermore, we have introduced a new type to the language, `iptr`, which is an integer type that is guaranteed to be able to fit an address[10]. The result of this effort is a more complete and accurate LLVM semantics in Coq, and demonstrates the applicability of our memory model for real, complex languages.

Previously, the VIR memory model was based upon the CompCert and quasi-concrete semantics, so it suffered from the deficiencies with respect to optimization correctness mentioned in Section 2. The quasi-concrete layout meant that finite memory could not be accurately modeled, and casts from pointers to integers would impact the concrete memory layout, making these casts effectful computations, which could not be trivially removed from program, even when dead. Integrating our memory model rectifies these problems, and our handling of symbolic bytes also allows us to handle how LLVM's complex `undef` values interact with memory more faithfully.

This section will necessarily discuss VIR and the LLVM IR in some detail, but it is worth noting that our memory model is not specific to LLVM. The infinite and finite models above provide general semantics that are parameterized by addresses and symbolic bytes, as these parameters could vary depending on the programming language. This section describes the VIR instantiation of the framework and, along the way, addresses some challenges of formalizing LLVM IR semantics. As in the general framework, this instantiation yields both an *infinite memory* and a *finite memory*

[9] We delay this description to Section 4.2.2.

[10] The `iptr` type does not currently exist in the LLVM IR, but we propose adding it to the IR in order to better support casts from pointers to integers. It is analogous to the **`intptr_t`** type that exists in C.

version of the VIR semantics. Most of the development is parametric with respect to that choice, but we differentiate them as VIR^{inf} and VIR^{fin} where necessary.

4.1 Layered Interpreters

VIR is structured as a series of *layered interpreters*, as shown in Figure 7, each of which specifies some aspects of the LLVM IR semantics. These interpreters are built on top of *interaction trees* [36, 38], which are a Coq datatype of potentially infinite trees (used to model diverging programs) whose nodes are *uninterpreted* events, indicated by E_0–E_5 in the Figure. Each layer *handles* some subset of the events, defining their semantics, and leaving the rest for later layers to handle.

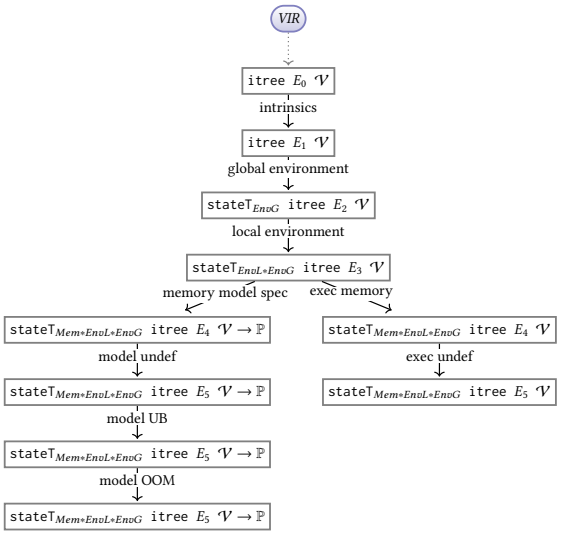

Fig. 7. Levels of interpretation. Each box shows the type of the semantic definitions at that layer. Arrows are labeled with the events they define.

For the purposes of this paper, the most important parts of the interpretation stack are the handlers for *memory events*, *nondeterminism*, *undefined behavior*, and *out-of-memory* exceptions. The memory events, $\mathcal{M}$, correspond to LLVM IR operations that interact with the memory model and each of those events is parameterized by appropriate input values and a return type $\mathcal{V}$ or $\mathcal{V}_u$ (indicated as a superscript), as shown below. These types, describing *dynamic values*, are explained in the next subsection. The other kinds of events are similarly annotated.[11]

Memory model interaction events

$$\mathcal{M} \triangleq \texttt{MPush}^{()} \mid \texttt{MPop}^{()} \mid \texttt{Load}^{\mathcal{V}_u}(\tau, a) \mid \texttt{Store}^{()}(a, uv) \mid \texttt{Alloca}^{\mathcal{V}}(\tau)$$

Undefined Behavior event

$$\mathcal{U} \triangleq \texttt{UB}^{\emptyset}$$

Nondeterminism events

$$\mathcal{N} \triangleq \texttt{Pick}^{\mathcal{V}}(uv) \mid \texttt{PickUnique}^{\mathcal{V}}(uv) \mid \texttt{PickNonPoison}^{\mathcal{V}}(uv)$$

OOM event

$$\mathcal{O} \triangleq \texttt{O}^{\emptyset}$$

The memory model is integrated with the VIR semantics via the handler for the $\mathcal{M}$ events, which is implemented in terms of the primitive operations described in Section 3. That handler is plugged into the stack of Figure 7 to interpret $\mathcal{M} \in E_3$ into stateful operations that manipulate the memory. As shown in the left-hand-path of the interpretation stack, the *specification* of the memory model is defined propositionally in Coq to account for the nondeterminism of possible implementations, including nondeterminism introduced by the memory model. The right-hand path implements the *executable* version of the semantics. On the specification side, the handlers for $\mathcal{U}$, $\mathcal{N}$ and $\mathcal{O}$ events interact with the notion of *refinement* for the behaviors of VIR programs (see Section 4.4). There are no equivalent handlers for $\mathcal{U}$ and $\mathcal{O}$ on the executable side, because the OCaml framework for executing VIR ITrees simply raises an exception when those events are encountered.

[11] Note that the superscript $\emptyset$ for $\texttt{UB}^{\emptyset}$ and $\texttt{O}^{\emptyset}$ means that these events do not return any value; they simply terminate the computation. The $\texttt{PickUnique}^{\mathcal{V}}(uv)$ and $\texttt{PickNonPoison}^{\mathcal{V}}(uv)$ events technically return *constrained* types that guarantee uniqueness or a non-poison value, respectively, but we elide those details here.

If p is a VIR program, we write, e.g., $[\![p]\!]_3$ to mean interpretation down to the 3^{rd} layer (i.e., just before the split). We write $[\![p]\!]_{\text{VIR}}$ for the "fully interpreted" semantics of p, i.e., at the bottom of the left branch, and we write `interpret` p for the executable version, at the bottom right branch. That the executable semantics is a valid implementation of the specification is one of the main theorems about the VIR development (see Theorem 6.1).

4.2 VIR Values

4.2.1 Dynamic Values and `iptr`*.* The semantics of VIR relies upon the domain $\mathcal{V}$ of *dynamic values* that the language can manipulate. The core of these dynamic values are the *defined values*.

$$dv \in \mathcal{V} ::= \texttt{none} \mid a \mid [\texttt{list}\ \mathcal{V}] \mid i_1 \mid i_8 \mid i_{16} \mid i_{32} \mid i_{64} \mid i_{ptr} \mid \texttt{poison}_\tau$$

The void value, `none`, is a placeholder for operations with no meaningful return values. Memory addresses (a), of type *Addr*, are implemented either as positive integers, $Addr^{inf} = \mathbb{Z}$, or 64-bit values $Addr^{fin} = \texttt{i64}$, depending on whether we are instantiating the infinite or finite memory model. VIR supports all of LLVM IR's structured values, including records, but here we present only arrays, noted as $[dv_1, \ldots, dv_n]$.

VIR supports `i1`, `i8`, `i16`, `i32` and `i64`-bit integers[12] ranged over by i_1, i_8, etc., but it also includes integers of type `iptr`, which are in bijection with memory addresses[13], and ranged over by i_{ptr} . `iptr` has the same cardinality as the *Addr* type, i.e., $\texttt{iptr}^{inf} \triangleq \mathbb{Z}$ and $\texttt{iptr}^{fin} \triangleq i64$. The `iptr` type acts mostly like the other integer types, supporting all of the same instructions, which allows for programs to perform arbitrary arithmetic on physical addresses *without* forcing a cast to a type of a fixed finite size. Because `iptr` has the same cardinality as *Addr*, pointer–integer casts can effectively be pure no-ops, allowing these casts to be removed to make way for other optimizations, and ensuring a round-trip property where a pointer cast to an integer and back yields the same pointer[14]. In order to preserve this round-trip property for casts, arithmetic on `iptr` values in VIR^{fin} may provoke *OOM*, as discussed in Section 4.3.3.

VIR also includes *poisoned values* (`poison`) representing a *deferred* undefined behavior [23]. Deferred UB is instrumental for aggressive optimizations, but a semantic subtlety. The `poison` value is a tainting mark: it propagates to all values that depend on it, so equivalences such as $\texttt{poison} + \texttt{poison} \equiv 2 * \texttt{poison} \equiv \texttt{poison}$ hold true.

4.2.2 Undef Values and Symbolic Bytes. LLVM represents uninitialized memory through *undefined values*, which represent the set of possible values of a given type, and operations on them manipulate a those sets. Reasoning about undefined values is subtle; each time an `undef` value is used within an LLVM program, it may take on a different concrete value. For instance, $\texttt{undef}_{i64} + \texttt{undef}_{i64} \equiv \texttt{undef}_{i64}$, whereas $\texttt{undef}_{i64} + \texttt{undef}_{i64} \not\equiv 2 * \texttt{undef}_{i64}$, because the value on the right hand side of the equation cannot be an odd number.

In VIR, "*uvalues*" or *under-defined values*, written $uv : \mathcal{V}_u$, model these sets. They are given by:

$$\begin{aligned} uv \in \mathcal{V}_u &::= \uparrow \mathcal{V} \mid [\texttt{list}\ \mathcal{V}_u] \mid \texttt{undef}_\tau \mid \texttt{op}\ \mathcal{V}_u\ \mathcal{V}_u \mid [\texttt{list}\ \mathcal{SB}]_\tau \\ sb \in \mathcal{SB} &::= \texttt{sbyte}_\tau(uv, i, sid) \end{aligned}$$

[12]We use CompCert's finite integers in our development. VIR also supports floating-point values (omitted here for simplicity).
[13]The `iptr` type is missing from the LLVM IR, but there is precedent for it in C, via **`intptr_t`**, **`uintptr_t`** and **`size_t`** [14]
[14]Note that there are concerns about provenance for such casts that we do not tackle in this paper [25]. Our implementation does not track provenance through integers, so our integer–pointer casts will always yield a pointer with a wildcard provenance, which is safe, but can limit optimizations in rare cases (see the discussion in [31]). This provenance tracking is largely orthogonal to the design of our memory model, however, our memory model can be integrated with a language regardless of whether it tracks provenance through integer computations.

```
%x = i16 select i1 undef, i16 0x1234, i16 0x0000
%y = i16 select i1 undef, i16 0x5678, i16 0x0000
store i16 %x, ptr %ptr
store i16 %y, ptr (inttoptr (add iptr (ptrtoint %ptr) 2)
%z = load i32, ptr %ptr
```

Fig. 8. Entangled bytes. Here the `select` instruction chooses between two values nondeterministically due to `undef`, but the `load` should read (assuming big-endianess for simplicity) one of `0x00000000`, `0x00005678`, `0x12340000`, or `0x12345678`, but never something like `0x00345600`.

Under-defined values include defined (i.e. concrete) values—we write $\uparrow$ for the corresponding injection—as well as *arrays* of uvalues. They also include $\texttt{undef}_\tau$, which stands for the *set* of all possible concrete values of the type τ (we omit τ when it is unimportant). $\mathcal{V}_u$ also includes "delayed" operations, where `op` ranges over VIR's arithmetic, bit-logic, and other computation primitives. Such a uvalue lifts the set semantics of `undef` to nondeterministic computations, as we explain below. The uvalues also contain (type-annotated) concatenations of *symbolic bytes* [5], $sb : \mathcal{SB}$, explained next.

Serializing under-defined values. Values in VIR are stored in memory as lists of bytes. An undefined value is serialized as symbolic bytes, given by the type $\mathcal{SB}$. A $\texttt{sbyte}_\tau(uv, i, sid)$ represents the i^{th} byte of the value uv with a store-ID sid, whereas $[\texttt{list}\ \mathcal{SB}]_\tau$ concatenates a series of symbolic bytes into a under-defined value of type τ.

To enable optimizations like store-forwarding, the semantics must precisely preserve nondeterminism when serializing and deserializing under-defined values to and from bytes. For example, the event handler for `Store` serializes a $uv \in \mathcal{V}_u$ into an array of symbolic bytes matching the size of the type (written $|\tau|$), with each byte containing the appropriate index into the under-defined value. This serialization operation is defined as:

$$\begin{array}{lcl} \texttt{serialize}(uv, \tau) & = & sid \leftarrow \texttt{fresh\_sid};; \\ & & \texttt{ret}\ [\texttt{sbyte}_\tau(uv, 0, sid), \ldots, \texttt{sbyte}_\tau(uv, |\tau|, sid)]_\tau \end{array}$$

Each of these symbolic bytes contains that same "store id" (sid), which is uniquely generated for every `Store` event, to preserve "entangled" `undef` values within the semantics. This sid is assigned to the serialized bytes and it is used to prevent the introduction of too much nondeterminism when reading bytes written by multiple stores. For example, the program in Figure 8 illustrates the scenario in which the first two bytes and final two bytes of `z` are entangled together, so there are only two possible values for each of these two byte chunks. Note that, if two symbolic bytes have the same sid, they must have come from the same store, and thus agree on their underlying uv too.

Conversely, the handler for the `Load` instruction *deserializes* the symbolic bytes:

$$\begin{array}{l} \texttt{deserialize}([sb_1, \ldots, sb_n]_\tau, \tau') = \\ \texttt{if}\ \tau = \tau' \ \wedge\ [sb_1, \ldots, sb_n]_\tau = [\texttt{sbyte}_\tau(uv, 0, sid), \ldots, \texttt{sbyte}_\tau(uv, |\tau|, sid)]_\tau\ \texttt{then}\ uv\ \texttt{else}\ [sb_1, \ldots, sb_n]_{\tau'} \end{array}$$

In the simple case where the value is loaded with the same type, the deserialization for `Load` can simply extract the original uv[15] (a property which makes store forwarding optimizations easy to justify). In more complex cases, where bytes are read at a different type from what they are stored at (possibly reading a portion of the bytes from different under-defined values) the resulting under-defined value is left as the concatenation of symbolic bytes but with the updated type—their concrete bit patterns will be resolved via "concretization," as explained next.

[15]There is a corner case for types where $|\tau| = 0$ (such as an array of length zero); in that case the correct uv is uniquely determined by τ.

4.3 Concretization: Refinement and Evaluation of LLVM Values

As we saw above, an under-defined value uv denotes a *set* of "concrete" dynamic values—that is, it is a *specification* of the set of allowed bit patterns a compliant implementation can use to refine uv. We write $[\![-]\!]_C : \mathcal{V}_u \to \mathcal{P}(Result(\mathcal{V}))$ to denote the (monadic) function that computes a set of concretizations of uv. This set of concrete values is the semantic meaning of uv, making concretization an important aspect of the semantics, allowing programs with different $\mathcal{V}_u$ representations for the same set to be compared. We write $dv \in [\![uv]\!]_C$, defined as $[\![uv]\!]_C \ni ok(dv)$ to indicate that (concrete) dynamic value dv is a legal *refinement* of uv. The concretization function is implemented in a similar fashion to the prior VIR semantics [39]. It essentially implements an interpreter for all of the computational instructions "lifting" them to work on sets of values. One important base case is that $\texttt{undef}_\tau$ concretizes to the set of all legal values of type τ, that is: $[\![\texttt{undef}_\tau]\!]_C = \{dv | dv : \tau, dv \neq \texttt{poison}\}$. For example, we have $\texttt{2} \in [\![\texttt{mul i64 2, i64 1}]\!]_C$ and also $\texttt{2} \in [\![\texttt{mul i64 1, undef}_{\texttt{i64}}]\!]_C$ but $\texttt{2} \notin [\![\texttt{mul i64 3, undef}_{\texttt{i64}}]\!]_C$. As you can see, due to the presence of arithmetic (and other non-trivial) LLVM IR operations, and the fact that under-defined values *include* ordinary values as a special case, the refinement relationship acts as an *evaluation* relation—in the case that uv has no occurrences of `undef` (i.e., it is defined), then $dv \in [\![uv]\!]_C$ simply means that uv evaluates to dv according to the ordinary rules of LLVM IR computations, as in the first example above.

Note that concretization can fail with *UB* (in case of, for example, division by `0`) or with *OOM* (when working with `iptr` values, as described below).

4.3.1 Concretizing Symbolic Bytes. New to this work is the treatment of symbolic bytes. Recall that symbolic bytes represent byte-sized fragments of (possibly) under-defined values and that a `Load` event might read a sequence of such bytes that were written by (several) different `Stores`. Series of symbolic bytes are concretized as shown below:

$$
\begin{aligned}
&[\![[\texttt{sbyte}_{\tau_1}(uv_1, 0, sid_1), \ldots, \texttt{sbyte}_{\tau_n}(uv_n, n, sid_n)]_\tau]\!]_C = \\
&\quad dv_1 \leftarrow [\![uv_1]\!]_C \;;;\; \ldots \;;;\; dv_n \leftarrow [\![uv_n]\!]_C \;;; \\
&\quad \{\texttt{ok}(\texttt{bitcast}_\tau(dv_1[1]dv_2[2]\ldots dv_n[n])) \mid \forall jk,\ sid_j = sid_k \to dv_j = dv_k\}
\end{aligned}
$$

This works by recursively concretizing the uv_i's, each of which yields a set of concrete dynamic values $[\![uv_i]\!]_C$ (or an error, in which case the whole concretization is an error). Then, from each $dv_i \in [\![uv_i]\!]_C$ we can extract the (concrete) i^{th} byte, written as $dv_i[i]$. The resulting set of concrete values is then obtained by concatenating the individual bytes combinatorially; however, if two symbolic bytes share a *sid* (and hence come from the *same* `store`) they are "entangled" and must be concretized in the same way.

The resulting sequence of bytes is converted to a dynamic value of type τ by the $\texttt{bitcast}_\tau(-)$ operation. It might need to truncate or pad the sequence, depending on the size of values of type τ and the number, n, of available bytes. This `bitcast` operation is, ultimately, what allows the conversion between values of distinct types. For instance, it could convert an array value of type $[8 \times \texttt{i8}]$ into a value of type `i64`, even though such values have different representations in the semantics. Altogether, this treatment of symbolic bytes properly ensures "entanglement" of values as illustrated in Figure 8.

4.3.2 Concretizing $\texttt{iptr}^{inf}$ *Values.* For the infinite memory model, in which `iptr` is taken to be (unbounded) integers, concretization works in essentially the same fashion as the other integer types. This is the easy case, because in VIR^{inf} `iptr` values are just $\mathbb{Z}$ values, so no overflow or underflow can occur, and all of the operations work straightforwardly as expected.

4.3.3 Concretizing $\texttt{iptr}^{fin}$ *Values.* The finite memory model defines `iptr` to be unsigned 64-bit integers. However, unlike for ordinary `i64` arithmetic operations, in which LLVM IR's `nuw` and `nsw` ("No un/signed wrap") flags cause overlow/underflow to be treated as introducing undefined behavior, for `iptr`, such errors instead introduce *OOM*. This difference from "ordinary" integers is crucial to maintaining the connection between the infinite and finite semantics. To see why, consider the following program (written using C-like notation that is easy to express as VIR code):

```
iptr i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

In the infinite language the `iptr` addition can never overflow, so this program will count up indefinitely, never calling the `do_evil` function. If we naïvely "convert" this program to a finite program by simply using 64-bit arithmetic, which can wrap, the value of `i` will overflow to the value `0`, terminating the loop and thus calling `do_evil`, which is not an allowed refinement. From this example we can see that it's clearly *not* safe to allow $\texttt{iptr}^{fin}$ values to wrap in general, as that can change the meaning of the program. LLVM's `nuw` flag also does the wrong thing—it introduces undefined behavior, so translating the infinite program to a finite program in this way would cause the target program to have UB while the source program does not!

Ultimately, the only reasonable solution is to add bounds checks to integer operations on `iptr` values and to halt the program with *OOM* when the checks fail (intuitively, such an arithmetic operation has run out of bits in which to store the result). The VIR^{fin} semantics incorporates these bounds checks directly into the specification of arithmetic on `iptr` values, as part of concretization, but these bounds checks could be added explicitly on top of regular `i64` values if desired.

4.4 Behavioral Refinement Within VIR^{X}

If we fix our attention on just one of VIR^{inf} or VIR^{fin}—call it VIR^{X}—and consider the interpretation stacks as shown in Figure 7, there are several notions of *behavioral refinement* that are useful for reasoning about the semantics. First, there is refinement at each successive layer of interpretation—that is, we can think of interpretation down to each layer as defining a program semantics with its own notion of refinement. Following [39], a key result about of the VIR development shows that refinement at one layer *implies* refinement at the next layer, which allows reasoning at one stage of the interpretation stack to be used to prove results about the "full" semantic interpretation.

Up until the interpretation of memory events, refinement is built on stateful variants of the $\texttt{eutt}_R$ bisimulation relation as defined previously [39]. For instance, after interpreting the local and global environments at layer 2, we would have the following top-level refinement relation between the behaviors of programs under a given environment represented by the ITrees P and Q of type $\texttt{itree}\ E_2\ (EnvL \times (EnvG \times \mathcal{V}))$:

$$P \sqsupseteq_2 Q \quad := \quad \texttt{eutt}_{\approx_{env_2}}(P,\ Q)$$

Typically, we instantiate the refinement relation by using it on the interpretations of program *syntax*, i.e., by taking $P = [\![p]\!]_2\ g\ l$ and $Q = [\![q]\!]_2\ g\ l$, where g and l are global and local environments. The relation $\approx_{env_2}$ acts as a postcondition on the results computed by the P and Q; in this case, it states that the returned values are equivalent, ignoring the global and local environments.

Once memory events are interpreted the semantics is nondeterministic, as the handler for $\mathcal{M}$ events (which defines the meanings of `MPush`, `MPop`, `Load`, `Store`, and `Alloca`) is implemented using the nondeterministic primitives of the general memory model framework operations from Figure 2, along with the `serialize` and `deserialize` mechanisms described above. The interpretation of $\mathcal{N}$ events also introduces further nondeterminism due to the treatment of `undef` values. The

refinement relation after interpreting memory and nondeterminism events is given by a set inclusion relation between sets of ITrees P and Q of type $\mathcal{P}(\texttt{itree}\ E_4\ Mem \times (EnvL \times (EnvG \times \mathcal{V})))$:

$$P \sqsupseteq_4 Q \quad := \quad \forall t' \in Q, \exists t \in P, \texttt{eutt}_{\approx_{env_4}}(t, t')$$

The sets of ITrees are generally taken to be those given by the interpretation of program syntax using the propositional semantics, so $P = [\![p]\!]_4\ g\ l\ sid\ m$ and $Q = [\![q]\!]_4\ g\ l\ sid\ m$, where g and l are the initial global and local environments as before, *sid* is the initial high watermark for store ids, and m is the initial state of the memory.

Finally, we would like to take UB and OOM into account. The semantics of $\texttt{UB}^{\emptyset}$ provides "time traveling" undefined behavior semantics [11]. Intuitively, any program, here represented as an interaction tree, that may reach a $\texttt{UB}^{\emptyset}$ event is considered to be ill-defined. We write this predicate as $\texttt{hasUB}(t)$, and, in that case, *any* other behavior is allowed in its set of refinements. Dually, the treatment of O events says that an *out-of-memory* event refines *any* behavior (but not in a "time-traveling" fashion—the programs must agree up until the $\texttt{O}^{\emptyset}$ occurs). That notion is defined via a modified version of the ordinary $\texttt{eutt}_R$ relation, which we write as $\texttt{eutt\_oom}_R$. Like eutt, eutt_oom is a weak simulation relation, but it additionally allows $\texttt{eutt\_oom}_R(t, \texttt{trigger}\ \texttt{O}^{\emptyset})$ for *any* interaction tree t—this is the sense in which "out-of-memory" refines everything.

Taken altogether, these definitions lead to the following top-level, definition of semantic refinement for two sets of behaviors P and Q:

$$P \sqsupseteq_{\text{VIR}} Q \quad := \quad \forall t' \in Q, \exists t \in P, \texttt{hasUB}(t) \ \vee\ \texttt{eutt\_oom}_{\approx_{env_{\text{VIR}}}}(t, t')$$

Once again, we can define refinement for VIR programs p and q by taking $P = [\![p]\!]_{\text{VIR}}\ g\ l\ sid\ m$ and $Q = [\![q]\!]_{\text{VIR}}\ g\ l\ sid\ m$.

4.4.1 Refinement Theorems for VIR^X. As mentioned above, the VIR development proves that refinement at lower levels in the interpretation stack of Figure 7 imply refinement at later levels (these are, intuitively, easy to prove because the less interpretation that has been done, the *stronger* the notion of refinement is). That means we can prove:

Theorem 4.1 (Level refinement). *For interpretations levels $\ell \leq \ell'$ and for any behaviors P and Q, if $P \sqsupseteq_\ell Q$ then $P \sqsupseteq_{\ell'} Q$. In particular, for any ℓ, we have $P \sqsupseteq_\ell Q$ implies $P \sqsupseteq_{\text{VIR}} Q$.*

Equally important is the ability to serially compose program refinements *within* a level of interpretation. This is needed to prove a pipeline of program optimizations correct. To this end, we prove:

Theorem 4.2 (Transitivity of refinement). *At every level ℓ, if $P \sqsupseteq_\ell Q$ and $Q \sqsupseteq_\ell R$ then it is also the case that $P \sqsupseteq_\ell R$.*

4.5 Lowering VIR^{inf} to VIR^{fin}

The main idea in this paper is to separate compilation into two distinct phases—there is an explicit transition from a source language such as VIR^{inf}, with semantics using infinite memory, to a "target" language such as VIR^{fin}, which uses a finite memory. Intuitively, when we convert an infinite program to a finite program the *only* difference in their behavior should be that the finite program can halt with an out-of-memory event at any point, instead of continuing execution. The semantics of VIR^{fin} is more constrained than that of VIR^{inf} because the finite address size and iptr size means that programs which allocate too much memory or compute addresses outside of the bounds of the finite memory cannot continue execution and must halt and trigger *OOM* instead. While our discussion in this section is centered around VIR, a similar structure of refinements would be used for other languages using our two-phase memory model.

We can express this connection as (yet another!) refinement. This relation is defined in terms of $\texttt{orutt}_R$ (similar to how $\texttt{eutt\_oom}_R$ is used to define the single-language refinements in Section 4.4). $\texttt{orutt}_R$ is a heterogeneous version of $\texttt{eutt\_oom}_R$, based on the $\texttt{rutt}_R$ relation between ITrees with different event structures instead of $\texttt{eutt}_R$, which operates on ITrees with the same event types. This is necessary as VIR^{inf} and VIR^{fin} have events which are parameterized by the types of addresses and `iptr` values.

The correspondence between memory configurations is given by the (overloaded) $\gtrsim$ relation shown in Figure 6. To express the relationship between VIR^{inf} under-defined values and VIR^{fin} ones, we also need to instantiate the `lift_sbyte` function required in that Figure. To do so, we simply lift the $\lceil p \rceil$ operation (which injects pointers) to all of the *uv* cases—the resulting relation is an injection that lifts a finite *uv* to its infinite counterpart, $\lceil uv \rceil$. That definition allow us to define the $\gtrsim$ relation for environments too.

Putting all the pieces together, yields the following definition:

$$P \gtrsim_{\text{VIR}} Q := \quad \forall t' \in Q, \exists t \in P.\ \texttt{hasUB}(t)\ \vee\ \texttt{orutt}_{(\gtrsim_{mem} \otimes \gtrsim_{env})}(t, t')$$

This definition says that for every behavior exhibited by the finite semantics, Q (as represented by the ITree t'), we can find a corresponding behavior, t' in the infinite semantics, P. The ITrees that represent the behaviors should agree with each other, either continuing indefinitely, or until both ITrees terminate in lock-step (by raising an error or returning a value successfully), or until the finite ITree raises an out-of-memory event. Finally this relation considers $\texttt{UB}^{\emptyset}$, if any ITree in P contains UB the relation holds.

Theorem 4.3 (Infinite-to-finite Top-level Refinement). *For every* VIR *program p,*

$$[\![p]\!]^{inf}_{\text{VIR}}\ g^{inf}_{init}\ l^{inf}_{init}\ sid^{inf}_{init}\ m^{inf}_{init} \gtrsim_{\text{VIR}} [\![p]\!]^{fin}_{\text{VIR}}\ g^{fin}_{init}\ l^{fin}_{init}\ sid^{fin}_{init}\ m^{fin}_{init}$$

This guarantees that our translation does not add any new behaviors, and that the finite program will behave identically to the infinite one until the programs terminate in lock-step, or the finite program runs out of memory. Despite the apparent simplicity, this is a very technically challenging theorem to prove for several reasons. First, because it quantifies over *all* programs, it touches the full semantics of both VIR^{inf} and VIR^{fin}, which, for LLVM IR, involves dozens of arithmetic, bitwise, logic, and datatype manipulation instructions—there are literally hundreds of cases to consider.

Second, it is asking us to prove an *existential* claim. Digging into the proof, we end up needing a lemma roughly of the form:

$$\forall t^{inf}\, t^{fin}, \texttt{orutt}_{R_1}(t^{inf}, t^{fin}) \rightarrow \forall t_2^{fin} [\![t^{fin}]\!] \ni t_2^{fin} \rightarrow \exists t_2^{inf}, [\![t^{inf}]\!] \ni t_2^{inf}\ \wedge\ \texttt{orutt}_{R_2}(t_2^{inf}, t_2^{fin})$$

That is, we need to find a VIR^{inf} tree, t_2^{inf}, whose behaviors agree with the VIR^{fin} tree t_2^{fin} except for *OOM*. Ideally we would be able to use coinduction to walk through the $\texttt{orutt}_{R_1}(t^{inf}, t^{fin})$ relation to build t_2^{inf}, because that would give us the appropriate relationships between continuations nodes in corresponding parts of the ITrees. Unfortunately, existentials are *inductive* in Coq, so we cannot use coinduction to extract information from this relation until the existential is already instantiated... which is too late! We therefore have to define a coinductive function that lifts the finite t_2^{fin} to the infinite t_2^{inf}, and then re-derive the relationship between them.

Finally, because the semantic interpretations on both sides are defined by layers of monadic interpreters (as in Figure 7), the proof itself proceeds by establishing the connection between infinite and finite semantics at each layer, leading to many refinement lemmas, that together imply this theorem. (There are other technical hurdles too—the `orutt` relation used here and earlier is itself

```
define void @alloca_code() {
  %ptr = alloca i64
  ret void
}
```

```
define void @ptoi_code() {
  %ptr = alloca i64
  %i = ptrtoint ptr %ptr to iptr
  ret void
}
```

```
define void @ret_code() {
  ret void
}
```

Fig. 9. Example code for optimizations.

a non-trivial variant of the ITrees `rutt` mixed inductive-coinductive definition, which requires a significant amount of metatheory, for instance to prove transitivity, to be useful.)

5 Optimizations Under the Memory Model

This section explores some important program transformations enabled by our memory model using the code examples shown in Figure 9. We have verified refinement relations between these blocks of code, in both the infinite language and in the finite language (where applicable). Though we have not (yet) verified full-blown optimization passes based on these transformations[16], the semantic reasoning used in the following refinement proofs is representative of the key ideas needed for the general case. A notable aspect of these examples is that the infinite memory model allows for dead allocation removal while the finite memory model does not.

The main results, each verified in Coq, are as follows:

Optimization 1. Dead allocation removal (only allowed in the infinite model):

$$\forall g\ l\ sid\ m. [\![\texttt{@alloca\_code}]\!]_{\text{VIR}}\ g\ l\ sid\ m \supseteq_{\text{VIR}} [\![\texttt{@ret\_code}]\!]_{\text{VIR}}\ g\ l\ sid\ m$$

Note that the twin-allocation model and CompCertS models described in Section 2 are not able to perform this transformation in general, unless they can verify that the allocation always succeeds—otherwise, removing the allocation may cause the program to continue executing instead of halting. This is not a problem in our two-phase model because allocations in the infinitary semantics always succeed, so we never have to worry about failed allocations hiding extra behaviors of the program.

Optimization 2. Removing a `ptrtoint` cast (only allowed in infinite model):

$$\forall g\ l\ sid\ m. [\![\texttt{@ptoi\_code}]\!]_{\text{VIR}}\ g\ l\ sid\ m \supseteq_{\text{VIR}} [\![\texttt{@ret\_code}]\!]_{\text{VIR}}\ g\ l\ sid\ m$$

The twin-allocation and CompCertS models would be able to remove the `ptrtoint` cast in this example, but still would not be able to remove the `alloca` (as in the previous example). The quasi-concrete model cannot justify this refinement, because casting a pointer to an integer impacts the layout of the concrete memory and, in a finite setting, that could potentially result in the program halting (and thus removing the cast could change the behavior of the program). Again, this is something that the two-phase model is able to handle gracefully, as pointer to integer casts are essentially no-ops. The cast could be removed in both the finite and infinite models, but as per the previous example, the allocation can only be removed in the infinite.

Optimization 3. Adding an `alloca` (allowed in both the infinite and finite model):

$$\forall g\ l\ sid\ m. [\![\texttt{@ret\_code}]\!]_{\text{VIR}}\ g\ l\ sid\ m \supseteq_{\text{VIR}} [\![\texttt{@alloca\_code}]\!]_{\text{VIR}}\ g\ l\ sid\ m$$

Finally, we may wish to *add* an allocation to a program (certain optimizations may wish to cache a result, for instance). This proves tricky for the approach taken by CompCertS, which maintains an invariant that memory usage never increases after a program transformation. Both our infinite

[16] In general, doing that would require static analysis and non-trivial manipulation of VIR syntax, which, while certainly doable, is beyond the scope of this paper.

and finite models allow this, however, thanks to the out-of-memory refinement relations we've developed.

5.1 Bounds Checking Overhead

Our two-phase memory model ensures that pointer–integer casts never have an external effect, which allows them to be removed when performing program transformations. One might reasonably wonder, however, about the bounds checks on `iptr` arithmetic in VIR^{fin} and whether these would impact possible optimizations. They *do*, but we believe the impact should be fairly minimal for the following reasons.

Firstly, nearly all optimizations should be performed under VIR^{inf} semantics, prior to lowering the program into the finitary semantics. Under the infinitary semantics, `iptr` arithmetic is just arithmetic on $\mathbb{Z}$, and expressions involving `iptr` can be optimized in the infinite world using these unbounded integers as a model without bounds checks. Any `iptr` computations that happen to be dead can be removed prior to lowering the program into the finite world.

All normal optimizations can occur at the infinite level, and thus the *only* optimizations necessary to do on finite programs would involve removing the bounds checks required to trigger *OOM* that are added by the infinite to finite translation. These bounds checks can, naturally, have a performance impact; however, we believe that they will not be a significant impediment to the performance of real-world programs, and, in many cases, optimizations on finitary LLVM programs should be able to remove these bounds checks entirely. Consider the following possible use cases for `ptrtoint` casts `iptr` arithmetic, which cover many real-world use cases:

(1) Pointers cast to integers to use as a hash.
(2) XOR doubly-linked lists.
(3) Using the least-significant-bit of a pointer as a flag.
(4) Indexing into allocated blocks.

For (1), pointers can be cast to simple integer types, like `i64`, instead. The truncation does not matter in these use cases, as the program will not cast the value back to a pointer. This will, however, require programmers to make a choice to cast to the appropriate integer type.

Doubly-linked lists using xor (2) are an interesting use of pointer arithmetic, however the finite `iptr` values will be 64-bit values, and performing a bitwise xor cannot yield an out of bounds value. Similarly, bitwise operations that use unused bits in pointers as flags (3) cannot cause an overflow either, so bounds checking will not be necessary for these operations.

And, of course, another important case to consider is the use of `iptr` arithmetic to index into an allocated block. However, this use case should be covered by the LLVM IR's `getelementptr` operation instead, where bounds checks are unnecessary. If `getelementptr` is used to compute an out of bounds pointer, using that pointer to perform a memory access will cause $\text{UB}^{\emptyset}$ in the infinite semantics anyway due to mismatched provenances.

Finally, existing programming languages like Rust can achieve a great deal of performance, despite requiring bounds checking for array accesses [30]. We're optimistic that 1) most situations where `iptr` arithmetic will be used will fall into these cases and not require bounds checking, 2) in rarer circumstances, other LLVM optimizations may be able to remove the bounds checks, and 3) for any remaining bounds checks the costs will be minimal. We believe that the flexibility our memory model allows for optimizations prior to the finite language level will outweigh these rare costs

$$\begin{array}{rcl} \texttt{read}_b^{\texttt{run}}\ (p : Ptr) & : & MemExec(SByte) \\ \texttt{write}_b^{\texttt{run}}\ (p : Ptr)\ (b : SByte) & : & MemExec(unit) \\ \texttt{pushf}^{\texttt{run}} & : & MemExec(unit) \\ \texttt{popf}^{\texttt{run}} & : & MemExec(unit) \end{array} \qquad \begin{array}{rcl} \texttt{alloca}^{\texttt{run}}\ (\overrightarrow{b} : \texttt{list}\ SByte) & : & MemExec(Ptr) \\ \texttt{malloc}^{\texttt{run}}\ (\overrightarrow{b} : \texttt{list}\ SByte) & : & MemExec(Ptr) \\ \texttt{free}^{\texttt{run}}\ (p : Ptr) & : & MemExec(unit) \end{array}$$

Fig. 10. Executable memory model: low level operations

6 Executable Reference Interpreters

A formal specification of a language should be useful, in that it allows for validating optimizations of interest, but also faithful to existing implementations and informal specifications. Where usefulness is the realm of formal verification, faithfulness sends us back to a more traditional software engineering consideration: testing. This need for validation is well identified among contributors of formal semantics, and has even led to the development of dedicated tools and techniques to alleviate the pain: ad-hoc usage of big-step semantics [7, 10], the K framework [33], and skeletal semantics [8] all notably contribute in this direction.

6.1 Executable Memory Models

The ITree framework [36], on which we base our work, is extremely helpful for validating such large scale semantics as ITrees can be extracted to executable code. In our case, the memory model presented in Section 3 is not deterministic —a crucial necessity to faithfully characterize memory for LLVM. Therefore it's intrinsically non-executable, as we implement in Coq the specification monad propositionally, representing sets $\mathcal{P}(A)$ as predicates `A -> Prop`.

To facilitate testing (see below), we provide proven-correct, executable versions of the memory model. To lighten the induced development burden, we maintain the implementation as monadic code as parallel as possible to the specification, which helps, in particular, with mirroring of changes between them.

Figure 10 describes the executable memory model interface: it precisely mimics the specification, except that it lives in a *deterministic, executable monad*: $MemExec(X) \triangleq Conf \rightarrow Result(Conf \times X)$.

The implementations of each of these operations closely mirrors their specification counterparts. They syntactically diverge significantly only when the specification is nondeterministic, i.e., in the `fresh` and `find_bk` utilities needed for `alloca` and `malloc`.

On the executable side, `fresh` simply uses a trivial freshness monad, which increments a natural number to generate fresh provenance. Our current implementation of `find_bk` is currently quite elementary, but sufficient for our testing purpose: it looks up the largest addresses currently allocated, and returns the range of the required size of following addresses. More clever allocation strategies, such as those used by actual implementations of `malloc` to reduce memory fragmentation, could be implemented if relevant: the specification only enforces that the allocated block is contiguous, and disjoint from any other block.

6.1.1 Correctness of the executable memory models. We prove for each memory operation that its executable implementation is valid with respect to its specification counterpart. Since these implementations are pure Coq functions, validity is almost defined as point-wise set membership, ensuring that, for any initial state, the computed result belongs to the specification, or that the specification contains undefined behavior:

A basic memory model computation ($s : \mathit{MemExec}(X)$) is valid with respect to a specification ($\mathcal{S} : \mathit{MemSpec}(X)$) if:

$$\forall \sigma,\ (\mathcal{S}\ \sigma) \ni \mathit{UB} \vee (\mathcal{S}\ \sigma) \ni s\ \sigma$$

Our development proves these soundness lemmas for all of the memory model primitives.

6.2 Executable VIR

Section 4 describes the integration of our memory model into VIR, a formal model of LLVM IR. Figure 7 also shows the right-hand path of interpreters, which provide an executable implementation by specializing the *concretization* operation of Section 4.3 to pick default values for each $\texttt{undef}_\tau$ (for instance $\texttt{undef}_{\texttt{i8}}$ is `0`). Let us call the resulting top-level executable program $\texttt{interpret}_{\text{VIR}}$.

Using the soundness lemmas for the memory-model base operations, it is straightforward to show that the resulting deterministic interpretation function is a valid refinement of the semantics:

Theorem 6.1 (Interpreter is sound). *For all programs p,*

$$[\![p]\!]_{\text{VIR}}\ g_{init}\ l_{init}\ sid_{init}\ m_{init} \sqsupseteq_{\text{VIR}} \{\texttt{interpret}_{\text{VIR}}\ p\}\ g_{init}\ l_{init}\ sid_{init}\ m_{init}$$

That is, the (singleton set) of behaviors defined by the executable interpreter refines the semantic specification—in other words, the interpreter is "correct."

6.2.1 Testing the VIR *semantics.* The resulting VIR interpreter, even with the somewhat complex memory model that manipulates symbolic bytes, is performant enough to be able to run real LLVM IR code. We use it on a suite of test cases consisting of several hundred hand-written unit tests of LLVM IR semantic features, as well as on LLVM IR code generated by compiling source C programs. We have also experimented with using QuickChick [12] to randomly generate LLVM IR programs that stress-test the memory model, and we can use the ability to generate LLVM IR to instantiate parts of the Alive2 [29] suite as executable tests. In all cases, we do differential testing of the executable VIR model versus `llc` to look for problems on either side. In the process of developing the memory model for this project, such testing was invaluable to debugging the model. It also highlighted some ill-specified corner cases in the LLVM IR itself, for instance, it is unclear what the `getelementptr` instruction should do when computing addresses for structures and arrays whose data values are smaller than 8 bits and hence "share" an address in memory, and `extractelement` seems to have similar problems when vector elements are smaller than 8 bits, resulting in miscompilations.

7 Discussion

7.1 Additional related work

The individual phases in our two-phase memory model share a lot in common with the existing state of the art in memory models—especially those already discussed in detail in Section 2—but with the crucial distinction that our approach recognizes that the compilation pipeline for many programming languages involves a phase-change from higher level programs with unbounded memory semantics, to bounded machine code (a boundary which is awkwardly straddled by compiler IRs like LLVM, and lower level languages like C). Many projects have either an explicitly finite size of memory [5, 22, 28], or utilize a parameterized finite pointer type [19, 21]. C memory models [19, 21] often even have a `uintptr_t` type as a parameter, which is part of the inspiration for our `iptr` extension to LLVM. These works generally consider a single finite parameterization of their memory models, however, and do not relate different parameterizations of the memory models. This raises the question: how would the memory model with 32-bit pointers relate to its 64-bit parameterization? We provide the answer with our out-of-memory refinement relations, treating the unbounded specification as the ground truth, and finite parameterizations as refinements.

Our memory model is currently just a sequential one. Concurrent memory models [1, 9, 15, 17, 24, 28, 34, 35] are much more complex, but we believe the two-phase approach is orthogonal and would apply to concurrent models as well. There are also other considerations for undefined behavior in memory models, which we don't touch upon. In C, strict aliasing requirements are important for ruling out pointer aliasing via the types of pointers, which some memory models [19, 21] tackle. Languages like Rust have complex ownership rules for pointers that eliminate pointer aliasing at the type level, the semantics of which is tackled by the RustBelt [15] project.

7.2 The Two-Phase Memory Model in the Context of VIR

These improvements to the VIR semantics have been a substantial development effort, expanding the codebase by over three fold in terms of lines-of-Coq-code. We've aimed to keep things realistic while encapsulating the many complications present in a substantial subset of LLVM. For instance, `undef` is known to be incredibly complicated to reason about [23], and the under-defined values required to simulate `undef` contain over 30 constructors, making (proofs by) case analysis particularly arduous. Furthermore, `undef` interacts with the memory model and semantics in non-trivial ways, and many changes were made throughout the development to figure out precisely where under-defined values should undergo concretization and nondeterminism should be collapsed so as to enable as many optimizations in the semantics as possible. The nondeterminism in the specification monads has also been a challenge to work with, as illustrated in the discussion surrounding Theorem 4.3.

The product of this painstaking work is a parameterized semantics for a substantial LLVM-like language with an in-depth characterization of many intricate and interacting details like `undef`, undefined behavior, nondeterministic memory operations, and casts between pointers and integers. We have done so in an effort to ease justifying optimizations in a compiler, without the compiler itself having to maintain complicated invariants in order to prove the validity of important optimizations. Our verified two-phase compilation between memory models provides a novel approach to handling the complexities of low-level memory operations like casts between pointers and integers in the presence of high-level optimizations, and demonstrates the semantic necessity of considering finite memory when compiling programs to finite architectures, which is applicable to many languages.

Having put in this effort, we are now in position to reap many rewards. For instance the Helix project [40] is a verified compiler for a numerical programming language that targets VIR, and our interface should provide a more accurate view of LLVM memory which will lend more credence to the compilation pipeline for Helix. Similarly, our memory model should be amenable to separation logics built using Iris [16], which have been used in conjunction with VIR before [37, 38]. We believe that our richer memory model and higher-fidelity LLVM IR semantics will be a boon for these and future projects that depend upon VIR.

8 Data-Availability Statement

The development has been incorporated into the main Vellvm development, available on GitHub [41], and a snapshot of the source code, as well as a VM containing all of the dependencies, has been made available on Zenodo [2].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number 2247088.

References

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[2] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. *A Two-Phase Infinite/Finite Low-Level Memory Model*. https://doi.org/10.5281/zenodo.12518800

[3] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer International Publishing, Cham, 449–468. https://doi.org/10.1007/978-3-319-12736-1_24

[4] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 67–83. https://doi.org/10.1007/978-3-319-22102-1_5

[5] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: a memory-aware verified C compiler using a pointer as integer semantics. *Journal of Automated Reasoning* 63 (2019), 369–392. https://doi.org/10.1007/s10817-018-9496-y

[6] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. A verified CompCert front-end for a memory model supporting pointer arithmetic and uninitialised data. *Journal of Automated Reasoning* 62, 4 (2019), 433–480. https://doi.org/10.1007/s10817-017-9439-z

[7] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 87–100. https://doi.org/10.1145/2535838.2535876

[8] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

[9] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 100–110. https://doi.org/10.5555/3049832.3049844

[10] Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

[11] Raymond Chen. 2014. Undefined behavior can result in time travel (among other things, but time travel is the funkiest). https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633

[12] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*, Vol. 125. 126.

[13] Charles Ellison. 2012. *A Formal Semantics of C with Applications*. Ph. D. Dissertation. University of Illinois. https://doi.org/2142/34297

[14] ISO 9899:1999 1999. *Programming Languages — C*. Standard. International Organization for Standardization.

[15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

[16] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[18] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

[19] Robbert Krebbers. 2013. Aliasing Restrictions of C11 Formalized in Coq. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 50–65. https://doi.org/10.1007/978-3-319-03545-1_4

[20] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. 2014. Formal C Semantics: CompCert and the C Standard. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 543–548. https://doi.org/10.1007/978-3-319-08970-6_36

[21] Robbert Jan Krebbers. 2015. *The C standard formalized in Coq.* Ph. D. Dissertation. [Sl]:[Sn]. https://doi.org/2066/147182

[22] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (oct 2018), 28 pages. https://doi.org/10.1145/3276495

[23] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 633–647. https://doi.org/10.1145/3140587.3062343

[24] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

[25] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.* 6, POPL, Article 20 (jan 2022), 32 pages. https://doi.org/10.1145/3498681

[26] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning* 41, 1 (01 Jul 2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[27] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Toulouse, France. https://inria.hal.science/hal-01238879

[28] Liyi Li and Elsa Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *34rd European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[29] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[30] Alana Marzoev. 2022. How much does Rust's bounds checking actually cost? — blog.readyset.io. https://blog.readyset.io/bounds-checks/. [Accessed 27-02-2024].

[31] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (jan 2019), 32 pages. https://doi.org/10.1145/3290380

[32] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[33] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397 – 434. https://doi.org/10.1016/j.jlap.2010.03.012 Membrane computing and programming.

[34] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22. https://doi.org/10.1145/2487241.2487248

[35] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. *SIGPLAN Not.* 46, 1 (jan 2011), 43–54. https://doi.org/10.1145/1925844.1926393

[36] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (dec 2019), 32 pages. https://doi.org/10.1145/3371119

[37] Euisun Yoon. 2023. *Modular Semantics and Metatheory for LLVM IR.* Ph. D. Dissertation. University of Pennsylvania. https://doi.org/20.500.14332/59534

[38] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.* 6, ICFP, Article 99 (aug 2022), 29 pages. https://doi.org/10.1145/3547630

[39] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

[40] Vadim Zaliva, Ilia Zaichuk, and Franz Franchetti. 2020. Verified Translation Between Purely Functional and Imperative Domain Specific Languages in HELIX. In *Software Verification: 12th International Conference, VSTTE 2020, and 13th*

International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 33–49. https://doi.org/10.1007/978-3-030-63618-0_3

[41] Steve Zdancewic et al. [n. d.]. *Vellvm.* https://github.com/vellvm/vellvm

Received 2024-02-28; accepted 2024-06-18