Monadic interpreters for concurrent memory models

Executable semantics for a concurrent subset of LLVM IR

Nicolas Chappe
nicolas.chappe@ens-lyon.fr
Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP, UMR 5668
Lyon cedex 07, France

Ludovic Henrio
ludovic.henrio@cnrs.fr
CNRS, ENS de Lyon, Inria, Université Claude Bernard Lyon 1, LIP, UMR 5668
Lyon cedex 07, France

Yannick Zakowski
yannick.zakowski@inria.fr
Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP, UMR 5668
Lyon cedex 07, France

Abstract

Monadic interpreters have gained increasing attention as a powerful tool for modeling and reasoning about first order languages. In particular in the Coq ecosystem, the Choice Tree (CTree) library provides generic tools to craft such monadic interpreters in presence of divergence, stateful effects, failure, and nondeterminism. This monadic approach allows the definition of semantics for programming languages that are modular in its effects, compositional w.r.t. its syntax, and executable.

This paper demonstrates the use of CTrees to formalize a semantics for concurrency and weak memory models in Coq. We instantiate the approach by defining the semantics of a minimal concurrent subset of LLVM IR. Our semantics is built in successive stages, interpreting each aspect of the semantics separately. In particular, one stage encodes multithreading as an interleaving semantics, and another stage implements a weak memory model that supports various LLVM memory orderings. Furthermore, the modularity of the approach makes it possible to plug a different source language or memory model by changing a single interpretation phase. By leveraging the notions of (bi)similarity on CTrees, we establish the equational theory of our constructions, show how to transport equivalences through our layered construction, and prove simulation results between memory models. Finally, our model is executable, hence the semantics can be tested by extraction to OCaml.

Keywords: semantics, concurrency, Coq, LLVM

1 Introduction

In recent years, large-scale verification of industrial-strength software has become increasingly common [51] following the inspirational success of CompCert [38] in Coq, or CakeML [32] in Isabelle/HOL. However, such developments still require a tremendous amount of expertise and efforts. A significant body of work hence seeks to simplify this task, whether through richer semantic foundations [6, 11], or through richer proof principles [29, 54, 61].

In the Coq ecosystem, the *Interaction Trees* (ITree) library by Xia et al. [56, 57] has been influential over the recent years as a rich semantic toolbox for modeling first order languages. Inspired by advances in denotational semantics [8, 18, 47], the library provides an implementation of a coinductive variant of the freer monad [28]. This library provides access to monadic programming over symbolic events, tail recursive and general recursion, and interpretation of effects into monadic transformers in the style of one-shot algebraic effects. Concerning proofs, a rich theory of weak bisimilarity of computations enables both equational reasoning, and relational Hoare-style program logics. The approach has been used to model and verify a wide range of applications, such as networked servers [30, 60], transactional objects [39], noninterference [53], or memory-safe imperative programs [19].

But the largest application of the approach is arguably embodied by the Vellvm project. This project aims to formalize LLVM IR, the intermediate representation at the heart of the LLVM compilation infrastructure [35], and build verified tools upon it. LLVM IR is both the target language of a wide range of source languages, from C/C++ and Rust to Haskell, and an intermediate representation that targets most architectures. As such, investing effort into its verification is particularly worthwhile, as it takes part in the trusted codebase of an enormous range of projects. In a nutshell, the language itself is a low level language based on SSA-formed mutually recursive control flow graphs with a low level memory model.

While the Vellvm project takes its roots over a decade ago [62, 63], Zakowski et al. have restarted the project on denotational foundations using the ITree library [4, 58]. The approach has been coined by Zakowski et al. as *"a compositional, modular, and executable semantics"*. Compositional in that it is built by structural recursion on the syntax, and defines the meaning of open programs. Modular in that it defines and compose the semantics of each effect as independent handlers. Executable in that the model allows for the extraction of a verified executable interpreter suitable for testing.

Despite its success, the Vellvm project presents a major blind spot: it strictly restricts itself to sequential computations, ruling out entirely any modeling of concurrency. This shortcoming is particularly regrettable in that concurrency bugs are particularly difficult to detect by nature, being hard to reproduce trough testing. In this paper, we pave the road towards addressing this limitation. More specifically, we raise the following question: can a monadic model be built for a language such as LLVM IR in the presence of threads against a weak memory model? We answer positively by implementing one such model in Coq.

To achieve this result, we build on Chappe et al's recently introduced *Choice Trees* (CTrees) [10]. CTrees are a variant of ITrees, where the monad not only provides support for divergence, but also nondeterminism. Chappe et al. demonstrate how this is sufficient to build trace models for concurrency, illustrating the approach on CCS and a simple imperative language with cooperative scheduling.

To model concurrency in the context of LLVM IR, we provide the following contributions.

- We build a semantic model for a concurrent language (Section 3) by composing four passes: (1) representation into CTrees, (2) implementation of intra-thread effects, (3) interleaving of threads, and (4) implementation of inter-thread effects.
- We develop a meta-theory, showing in particular how equivalence of programs is transported across interpretation, which provides a simple proof method for a class of thread-local optimizations (Section 5).
- We apply our approach to μ_{IR}^{thread}, a simplified version of LLVM IR with support for thread creation (Section 3.2) and a weak memory model based on Kang et al's work on *Promising Semantics* [24] (Section 3.6).
- We derive an executable version of the semantics from our model (Section 4).

All our results are formalized in the Coq prover, and provided as an open source artifact.[1]

2 Context

2.1 Memory models and LLVM IR orderings

In a concurrent setting, the semantics of accesses to a shared memory can be particularly subtle. Indeed, modern architectures such as ARMv8 do not ensure that writes to memory are immediately visible to all threads, a property known as *sequential consistency* (SC). In turn, modern programming languages adopt memory models *weaker* than SC (i.e., allowing more behaviors) to enable efficient compilation to such targets. Otherwise, synchronization statements (e.g., fences) have to be injected by a compiler targeting hardware with a weaker memory model to ensure that the compilation does not introduce unexpected behaviors. These additional synchronizations induce a run-time performance penalty.

Intermediate representations for compilers such as LLVM IR are at the convergence of such constraints: they must support models allowing an efficient compilation both to a wide range of hardware, as well as from the vast majority of source languages. To accommodate for the diversity of front-ends it supports, LLVM IR's atomic memory access and fence instructions support a *memory ordering* annotation

```
        thread A
1  store monotonic 2, @x
2  fence release
```

```
        thread B
1  %1 = load @x
2  %2 = load monotonic @x
3  fence acquire
4  %3 = load @x
```

Listing 1. Fragment of an LLVM IR program with 2 threads running in parallel (simplified syntax).

that specifies the degree of atomicity of the instruction. We sum up their semantics below, and refer the interested reader to the LLVM language reference for further information.[2]

- Regular loads and stores, with no annotation,[3] offer little atomicity guarantees. They are unsafe in a concurrent setting, unless another form of synchronization such as fences or mutexes is used. In most cases, data races involving a non-atomic operation return an undefined value.
- The *Unordered* ordering corresponds to the Java memory model. It guarantees that a load returns a defined value that comes from a memory write to the same address, but it still offers little guarantee on which value is chosen.
- The *Monotonic* ordering corresponds to the relaxed C/C++ memory model. It enforces a total ordering on memory accesses to the same memory location, but not on those to different memory locations. For most weak hardware memory models, this ordering is the strongest one that can be efficiently compiled to machine code without introducing additional fences.
- The *Acquire*, *Release* and *AcquireRelease* orderings are based on their C/C++ counterparts. They offer synchronization guarantees on memory akin to mutexes. When an acquire operation synchronizes with a prior release operation (e.g., an *acquire read* reads a value from a *release write*), all the writes visible to the releasing thread become visible to the acquiring thread.
- *SequentiallyConsistent* is the strongest LLVM IR ordering. When used exclusively, it guarantees global sequential consistency.

Consider the litmus test in Listing 1 for illustration. Assuming @x is atomically initialized to 0, the non-atomic load of thread B (line B.1) may return undef as it can read both the initial 0 or the 2 from (A.1). By contrast, the monotonic load (B.2) will have a defined result, either 0 or 2, because it is atomic. Assuming the acquire fence (B.3) synchronizes with the release fence (A.2), all the stores visible to thread A at the time of the fence become visible to thread B, which implies that the final load at (B.4) unambiguously returns 2.

[1]See the supplementary material of this paper.

[2]https://llvm.org/docs/LangRef.html

[3]In μ_{IR}^{thread}, we use the annotation *not_atomic* for uniformity.

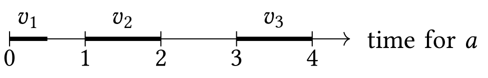


Figure 1. An example Promising timeline for address a

2.2 Promising Semantics

Capturing the semantics of weak memory models, or from our perspective of the various orderings we have informally described in the previous section, has constituted a major research endeavor over the last two decades. We seek a formal memory model that supports the different LLVM IR memory access operations (load, store, read-modify-write (RMW) and fence) and orderings. We furthermore need the model to be *operational*: by defining locally the next available transitions of the system, such models fit better in the CTree formalism. Promising Semantics [24] is one such operational weak memory model, and has been extensively studied over the past few years [13, 36, 37, 59]. We introduce its main features as it is a key source of inspiration for our memory model, but do not delve into the details of the associated meta-theory.

In its most basic form, Promising semantics uses two components to model a shared memory: a global set of *messages* and per-thread *views*. The set of *messages* materializes the past writes to memory.

Definition 2.1 (Promising Message)**.** A message $\langle addr := val@(start, end]\rangle$ represents a store of value val to address $addr$, happening at half-open time interval $(start, end]$, $start$ and end being rational timestamps in $\mathbb{Q}$.

Through the global set of messages, each address has a totally ordered timeline of its past stores. Two stores to the same address cannot have overlapping timestamps, and each memory address has its own *independent* clock, which means that $\langle a_1 := v_1@(1, 2]\rangle$ is guaranteed to happen before $\langle a_1 := v_2@(2, 3]\rangle$ but not necessarily before $\langle a_2 := v_3@(2, 3]\rangle$ because it involves a different address.

On top of the global set of messages, a Promising state also contains thread states. Each thread t has a *view* $view_t$ that records for each address the timestamp of its last access performed. When a message $\langle addr := val@(start, end]\rangle$ is read or written by some thread t, $view_t$ is updated so that $view_t(addr) = end$. The view of a given thread can only increase over time, but not necessarily to the maximal known timestamp.

Figure 1 represents the timeline for address a if the global message set contains the messages $\{m_1 = \langle a := v_1@(0, \frac{1}{2}]\rangle, m_2 = \langle a := v_2@(1, 2]\rangle, m_3 = \langle a := v_3@(3, 4]\}$.

The example in Listing 2, adapted from [24], demonstrates how timestamps enable store-store reordering in Promising semantics. a and b can both be assigned 1 in the same execution, if the store to y on line A.2 happens before (i.e., is assigned a lower timestamp than) the store to y on line B.1,

```
              thread A             ||               thread B
  1  store mon 2, @x ; t=2         ||   1  store mon 2, @y ; t=2
  2  store mon 1, @y ; t=1         ||   2  store mon 1, @x ; t=1
  3  %a = load @y    ; t=1         ||   3  %b = load @x    ; t=1
```

Listing 2. Fragment of an LLVM IR program with 2 threads (simplified syntax, `monotonic` abbreviated to `mon`). The comments indicate a possible assignment of timestamps at which load and store operations occur.

```
CoInductive ctree (E B : Type → Type) (R : Type) :=
(* pure computation *)
| Ret (r : R)
(* external event *)
| Vis {X : Type} (e : E X) (k : X → ctree)
(* delayed branching *)
| BrD {X : Type} (c : B X) (k : X → ctree)
(* stepping branching *)
| BrS {X : Type} (c : B X) (k : X → ctree)
```

Figure 2. The CTrees data structure

and similarly the store to x on line B.2 happens before the store to x on line A.1.

One may wonder why messages contain intervals instead of just one timestamp. The reason is that support for read-modify-write operations relies on these intervals: an interval can "fill" a timeline between the timestamp of the read part of an RMW operation and the timestamp of its write operation, in order to prevent the insertion of undesirable writes between them.

We stress that this short description of promising semantics is simplified. In particular, promising semantics also supports load-store reorderings thanks to *promises*. At any point of an execution, a thread can promise that it will later write some value to some address at some timestamp. Other threads accessing this address can read from this promise as if the future write had already happened. For an execution to be valid, every promise has to be eventually fulfilled. We will not support promises in our implementation, as extensively discussed in Section 3.6. This allows us to support all the orderings of an acquire/release semantics, but not the load-store reorderings allowed by the monotonic ordering.

2.3 Choice Trees

The datatype. CTrees, introduced in [10], is a Coq library providing a coinductive data-structure `ctree E B X` of potentially infinite trees. As illustrated in Figure 2, values of this datatype are built of four kinds of nodes. Leaves (`Ret`) carry values of type `X`: they represent pure computations. Computations can interact with the environment through external events (`Vis`) taken from the signature `E`. Finally, CTrees can perform two variants of nondeterministic branching (`BrD`

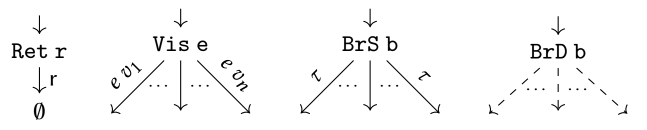


Figure 3. The interpretation of the four kinds of nodes of values of type `ctree E B X` in terms of a labeled transition system. Doted transition represents an inductive search down the tree for a transition.

and `BrS`) taken from the signature `B`.[4]. A signature is a family of types: for instance, an event `e : E nat` represents an interaction with the environment whose effect should return to the computation a natural number. Hence, a `Vis` node over `e` resumes with a continuation indexed over `nat`.

The semantic meaning of CTrees can be better understood through their propositional interpretation as Labeled Transition Systems (LTS). Figure 3 provides an informal definition of this LTS. Pure computations observe their resulting value r, and move to a stuck state, written ∅, and represented as a nullary `BrD` node. External events encode observable computations: each branch is labeled differently, by both the event performed and the answer subsequently received. Both branching nodes are internal actions, but are still further distinguished depending on their visibility. `BrS` nodes encode a computational step whose existence can be observed (denoted by the presence of a τ label in Figure 3). In contrast, `BrD` nodes are truly invisible, capturing a proper internal nondeterministic transition: the LTS can take a step if it finds inductively a step in any of the successors of the node. Observe in particular that an infinite tree made of `BrD` nodes constitutes a representation of the stuck process.

While the interface `B` is application dependent, we typically work with a baseline of branching choices `B01` allowing for representing stuck processes (∅, a `BrD` node with no successor), silent guards (`Guard`, a `BrD` node with a single successor), and stepping guards (`Step`, a `BrS` node with a single successor). We write `+'` for the disjoint sum of signatures.

Programming. Much like ITrees, CTrees come with a set of combinators that facilitates writing computations, or modeling source languages. First, `ctree E B` forms a monad for any interfaces `E` and `B`: they can embed pure computations and sequence computations through the `ret` and `bind` constructs. CTrees furthermore supports iteration, via the combinator `iter (f : I → ctrees E B (I + J)) (i : I)` that iterates recursively a loop body `f`, starting from the index `i`. If the body returns a new index `inl i'`, it loops; if it returns an exit signal `inr j`, it terminates. Of course arbitrary computations can always be defined directly by corecursion. Finally, CTrees support nondeterministic effects: the stuck process ∅ and the binary choice `choose p q := BrD T2 (| T21 ⇒ p | T22 ⇒ q)`[5] form the expected lattice; `choose` trivially generalizes to n-ary choice.

As a minimal illustration, one can model a little process randomly walking along a line with two events **`Right`**/**`Left`** `: Ewalk unit`:

```
CoFixpoint walk := choose3 (Vis Right (fun _ ⇒ walk))
                           (Vis Left  (fun _ ⇒ walk))
                           (Ret tt)
```

which models the two states LTS

$$\begin{tikzcd} p \arrow[loop above, "\mathtt{Right}"] \arrow[loop below, "\mathtt{Left}"] \arrow[r, "\mathtt{tt}"] & \emptyset \end{tikzcd}$$

Monadic interpretation of effects. As illustrated above, CTrees are a shallow embedding in Coq of possibly nondeterministic LTSs. As is, they are however quite limited for the purpose of modeling programming languages. Consider for instance an alternate version of `walk` that would at each iteration step twice to the right, once to the left, or vice-versa. We would like both processes to be equivalent! Furthermore, we may want to execute our process. In both case, external events are ill-fit.

To address this question, CTrees, following ITrees, support one-shot handlers of external events into monadic implementations. More specifically, the library provides a function `interp`. Given a handler `h: ∀ X, E X → M X` (abbreviated `E ⇝ M` hereafter), i.e., an implementation of the interface `E` into a monad `M`, `interp h: ctree E B ⇝ M` implements computations over `E` into `M`. The monad `M` must be able to internalize divergence, and non-determinism.

Coming back to `walk`, one may use a counter to keep track of the current coordinate and implement **`Right`** and **`Left`** as increment/decrement operations. The monad of implementation `M` would therefore be `Z → ctree ∅ B (Z * _)`, i.e., `stateT Z (ctree ∅ B)`. Given an initial coordinate, the computation does not contain any external event anymore: it is both suited for execution, and allows for proving that `walk` and its alternative are equivalent. In this paper, `M` will always be a stateful CTree monad of the form `stateT S (ctree E B)`.

Equivalence. Equivalence of CTrees is implemented as a standard notion of strong bisimilarity over their corresponding LTS (Figure 3). The inductive collapse of `BrD` nodes in the construction of the LTS before defining the bisimilarity is slightly reminiscent of the treatment of silent steps under

[4]The `B` parameter was not present in [10], branching was over finite sets `fin n`. We base our work on a later, more general version of the library.

[5]We write `T2` for a type with two inhabitants `T21` and `T22`, and abbreviate the notation for defining a function by case analysis.

weak bisimilarity: they are however different, we refer the interested reader to [10]. The library provides a rich equational library for its combinators w.r.t. strong bisimilarity.

When moving to richer monads through interpretation, one defines application specific equivalences. In the case of the stateful interpretation considered above, one could for instance work with pointwise bisimilarity.

Extraction. CTrees are a shallow model of LTSs, they do not rely on propositional relations. As such, they are perfectly extractible to OCaml, leading to executable reference interpreters. In particular, if no external event remains, they can be run against an implementation of a scheduler deciding how to choose how to crawl the nondeterministic branches.

3 Concurrent semantics for a subset of LLVM IR

This section introduces our approach to formalize concurrency and memory models as monadic interpreters. The approach is applied to a subset of LLVM IR focused on concurrency: an assembly-like language with concurrent memory accesses and functions that can be spawned with C-style thread creation and joining. However, the principles and the tools we develop are applicable to other languages and concurrency primitives.

In the remainder of this section, we first give a bird's eye view of our approach, before specifying the source language we consider, and defining its semantic model.

3.1 A semantic model built as an interpretation stack

Figure 4 illustrates the construction of the semantic model. It is structured into successive stages of interpretation, from a source language (μ_{IR}^{thread} in our case study, introduced hereafter) all the way down to our semantic domain $\mathcal{D}_4$, a monadic computation combining a read-only map of globals, stateful local and memory states, and internalizing the potential divergence and nondeterminism into a CTree. The stack follows four stages:

- *CTree representation.* From the source language, each function is represented into a (deterministic) CTree. This stage produces a list of CTrees.
- *Intra-thread interpretation.* This stage gives a semantics to thread-local events: this process can be done pointwise over $\mathcal{D}_1$, it is unrelated to the concurrent nature of the computation. For μ_{IR}^{thread}, this phase deals with accesses to globals and registers, introducing a reader monad (for global variables) and a state monad (for local variables) in $\mathcal{D}_2$.
- *Interleaving.* This pass takes the CTrees modeling the (spawnable) functions in $\mathcal{D}_2$, and builds a singular (nondeterministic) CTree that represents the concurrent execution of the program. Spawn events are given a semantics at this stage.
- *Inter-thread interpretation.* The final stage gives a semantics to the remaining, non-thread-local, events; in particular it interprets shared memory accesses. In our case-study, we build an operational Promising-like memory model supporting non-atomic, acquire/release, and (partly) monotonic accesses.

We emphasize that only the first layer of interpretation, the representation of the source language into $\mathcal{D}_1$, is language-specific. The other components of the model are reusable. Furthermore, alternate memory models can be plugged in place of the inter-thread interpretation; we come back to this idea in Section 5.3.

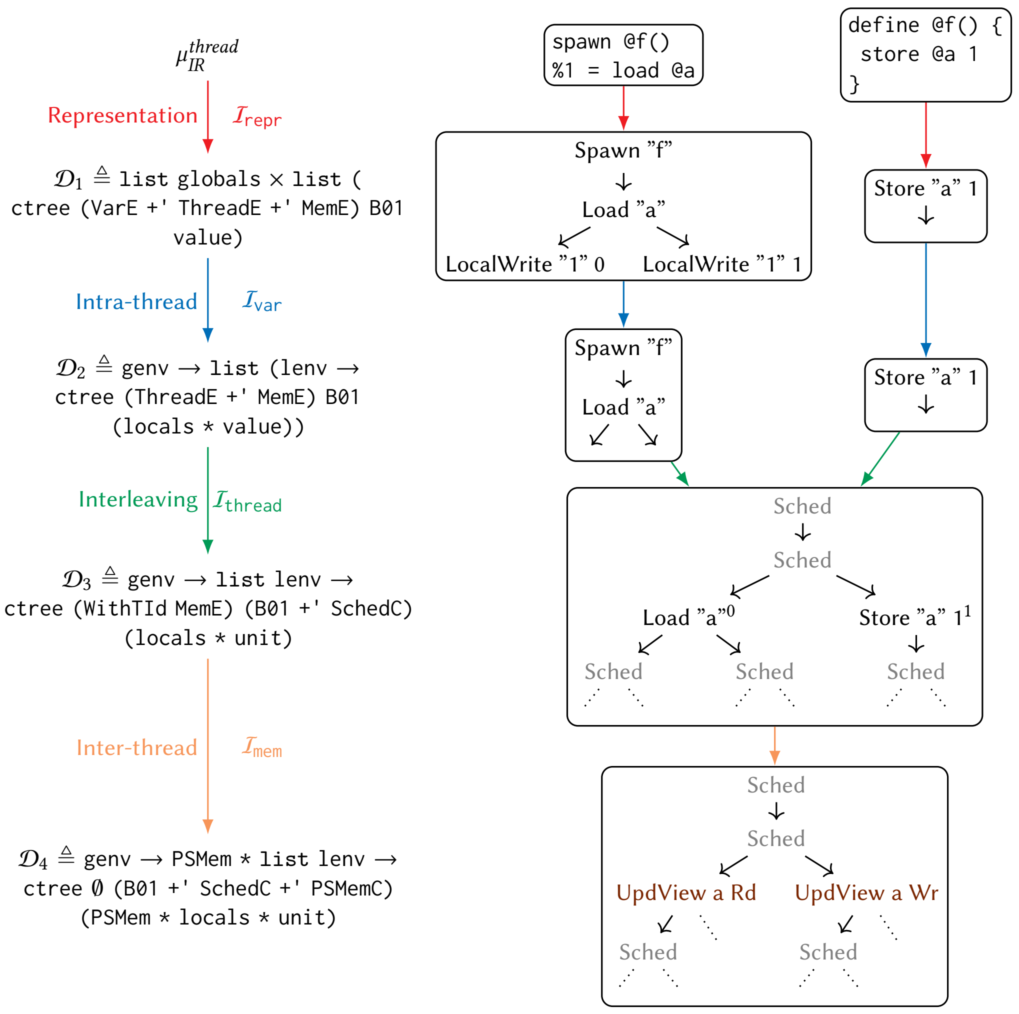
3.2 The source language: μ_{IR}^{thread}

Figure 5 depicts the syntax of μ_{IR}^{thread}, our source language. A program includes an identified *main* function, a list of global variable (@*id*) declarations, and a list of functions ready to be spawned. These functions take exactly one argument. They are defined as control flow graphs, i.e., a name, an entry block, and a list of blocks. Blocks contain an identifier, straight line three address code, and a terminator either returning, or jumping to a new block.

μ_{IR}^{thread} instructions include arithmetic operations (*exp*) and standard LLVM IR memory access instructions, annotated with their expected orderings, as described in Section 2.1. Since the semantics of thread creation is not defined in LLVM IR, and largely depends on the platform, language, and libraries used, we define a non-standard μ_{IR}^{thread} instruction `spawn` ($\mathit{fid}, \mathit{fid}_{\texttt{init}}, \mathit{fid}_{\texttt{cleanup}}$, x), that spawns a thread with the body of the function *fid* as its initial task, with x given as a parameter. This instruction is parameterized by a thread initialization function $\mathit{fid}_{\texttt{init}}$ and a thread cleanup function $\mathit{fid}_{\texttt{cleanup}}$, respectively run at the beginning and at the end of the thread execution.

Such a parameterized spawn primitive is very flexible. We leverage it to implement in μ_{IR}^{thread} thread creation and joining, based on the semantics of `thrd_create` and `thrd_join` from the C11 standard library [20]. Their semantics and our implementation rely on acquire/release accesses for synchronization. Due to the absence of function calls in μ_{IR}^{thread}, we use Coq-level macros to generate the code, but would use source-level functions in a more complete language like Vellvm. Appendix A provides additional details.

As any production level language, LLVM IR accumulates numerous orthogonal features, leading to active research even when restricted to its sequential memory model [4, 25]. In order to keep the complexity of our development reasonable, many LLVM IR features, mostly unrelated to concurrency concerns (typing, function calls other than via `spawn`, undefined behaviors, etc.) are not supported in our development. These excluded features are however supported in



μ_{IR}^{thread}
Representation $\mathcal{I}_{\text{repr}}$
$\mathcal{D}_1 \triangleq$ list globals × list (ctree (VarE +' ThreadE +' MemE) B01 value)
Intra-thread $\mathcal{I}_{\text{var}}$
$\mathcal{D}_2 \triangleq$ genv → list (lenv → ctree (ThreadE +' MemE) B01 (locals * value))
Interleaving $\mathcal{I}_{\text{thread}}$
$\mathcal{D}_3 \triangleq$ genv → list lenv → ctree (WithTId MemE) (B01 +' SchedC) (locals * unit)
Inter-thread $\mathcal{I}_{\text{mem}}$
$\mathcal{D}_4 \triangleq$ genv → PSMem * list lenv → ctree ∅ (B01 +' SchedC +' PSMemC) (PSMem * locals * unit)
spawn @f()
%1 = load @a
define @f() {
store @a 1
}
Spawn "f"
Load "a"
LocalWrite "1" 0
LocalWrite "1" 1
Store "a" 1
Sched
Load "a"^{0}
Store "a" 1^1
UpdView a Rd
UpdView a Wr

Figure 4. The interpretation stack: signatures (left) and simplified example (right). Black nodes represent events (`Vis`), brown ones stepping branches (`BrS`), and gray ones silent branches (`BrD`). `Ret` nodes are omitted, and dotted lines indicate further omitted nodes.

$$
\begin{array}{rcl}
\textit{atom} & ::= & @\textit{id} \mid \%\textit{id} \mid \textit{int} \mid \textit{bool} \mid \texttt{undef} \\
\textit{exp} & ::= & \textit{atom} \mid \textit{atom}\ \texttt{op}\ \textit{atom} \\
\textit{aop} & ::= & \textit{atomic\_exchange} \mid \textit{atomic\_add} \\
\textit{ord} & ::= & \textit{not\_atomic} \mid \textit{monotonic} \mid \textit{acquire} \mid \textit{release} \\
 & \mid & \textit{acq\_rel} \mid \textit{sc} \\
\textit{instr} & ::= & \textit{exp} \mid \texttt{alloca}\,(\textit{exp}) \mid \texttt{load}_{\textit{ord}}\,(\textit{exp}) \\
 & \mid & \texttt{store}_{\textit{ord}}\,(\textit{exp},\ \textit{exp}) \mid \texttt{rmw}_{\textit{ord}}\,(\textit{aop},\ \textit{exp},\ \textit{exp}) \\
 & \mid & \texttt{cmpxchg}_{\textit{ord}}\,(\textit{exp},\ \textit{exp},\ \textit{exp}) \mid \texttt{fence}_{\textit{ord}} \\
 & \mid & \texttt{spawn}\,(\textit{fid},\ \textit{fid},\ \textit{fid},\ x) \\
\textit{term} & ::= & \texttt{branch}\,(\textit{exp},\ \textit{bid},\ \textit{bid}) \mid \texttt{jmp}\,(\textit{bid}) \mid \texttt{return}\,(\textit{exp}) \\
\textit{sblock} & ::= & \{\textit{entry} : \textit{bid};\ \textit{code} : \textit{list}\ (\textit{fid}, \textit{instr});\ \textit{term} : \textit{term}\} \\
\textit{cfg} & ::= & \{\textit{name} : \textit{fid};\ \textit{entry} : \textit{id};\ \textit{body} : \textit{list}\ \textit{sblock}\} \\
\textit{prog} & ::= & \{\textit{main} : \textit{cfg};\ \textit{funs} : \textit{list}\ \textit{cfg};\ \textit{globs} : \textit{list}\ @\textit{id}\}
\end{array}
$$

Figure 5. Syntax for μ_{IR}^{thread}, a minimal subset of LLVM IR

Vellvm [58]. We expect that a future integration of our contributions to Vellvm would only require minor modifications to the way we handle concurrency and memory.

3.3 CTree representation for μ_{IR}^{thread}

This first step translates the syntax into the semantic domain $\mathcal{D}_1$: each function is denoted into a CTree, and collected into a list, along with the global variables. This process is rather standard, following closely Vellvm to resolve the control flow, albeit using CTrees rather than ITrees. In particular, graphs are denoted as a tail recursive fixpoint of the function mapping block identifiers to their denotation. We refer to Zakowski et al. [58] for details.

Crucial to this denotation is the identification of the effects of the language, captured for now into abstract events. We inventory them in Listing 3: interactions with the local and global variables (`VarE`), interactions with the shared memory

```
(* Events used in the initial representation *)
Variant VarE : Type → Type :=
| LocalWrite (id: ident) (v: value) : VarE unit
| LocalRead  (id: ident)            : VarE value
| GlobalRead (id: ident)            : VarE value

Variant MemE : Type → Type :=
| Read      (o: ordering) (k: addr)          : MemE value
| Write     o (k: addr) (v: value)           : MemE unit
| ReadWrite o (k: addr) (f: value → value) : MemE value
| Fence     (o: ordering)                    : MemE unit
| Alloc     (sz: nat)                        : MemE addr

Variant ThreadE : Type → Type :=
| Spawn (f init cleanup:fid) (arg:value): ThreadE thread_id
| Yield                                : ThreadE unit

(* Event and branch introduced in the interleaving *)
Variant WithTId (E : Type → Type) : Type → Type :=
| Annot {X} (e : E X) (t : thread_id) : WithTId E X

Variant SchedC : Type → Type :=
| Sched (ready: list thread_id) : SchedC thread_id

(* Additional branch introduced by the memory model *)
Variant PSAccess : Type :=
  PSRead | PSFulfill | PSFulfillUpdate
Variant PSMemC : Type → Type :=
| PSUpdateView (m: SMem) (i: thread_id) (a: addr) (acc:
     PSAccess): PSMemC (date * date)
```

Listing 3. Signature of events and branches used in the construction of the model

(`MemE`), and multithreading events (`ThreadE`). Note that these events only specify a signature at this stage: their semantics will be refined in the subsequent stages of interpretation; in particular, this leaves us all flexibility in choosing the memory model later on.

The intuitive semantics of variable and memory events is mostly straightforward. The most complex of these events is the RMW operation (`ReadWrite o k f`) that atomically reads a memory address `k`, modifies its content according to the function `f`, and returns the read value. Each memory event (save for `alloc`) takes a memory *ordering* as argument to specify atomicity constraints that the memory model should enforce on this access. These orderings directly reflect LLVM IR's specification, as discussed in Section 2.1.

`Yield` events are temporary placeholders adding synchronization points, which simplifies the operational characterization provided in Section 5.2. We add them to tag pure instructions and jumps between blocks. They are replaced by a `Guard` in the interleaving phase.

CTrees are not only parameterized by their return type and interface of events, but also by their interface of internal branching. The interface for internal branching in $\mathcal{D}_1$, `B01`, only allows for stuck branches and unary `Guard` constructors. Consequently, each function is modeled as a CTree with a single *deterministic* trace. The return type in $\mathcal{D}_1$ corresponds to the type of dynamic values. In μ_{IR}^{thread}, dynamic values are restricted to unbounded signed integers that also serve as pointers.

3.4 Interpretation of intra-thread events

By nature, the semantics of thread local events is orthogonal to any concurrency concern. We therefore handle them first, without introducing any observable event in the process—we come back to this intuition when characterizing our model operationally in Section 5.2. Note that in this second domain $\mathcal{D}_2$, the model of each function is still deterministic.

This interpretation pass is simple enough to be defined in terms of the generic `interp` combinator from the CTree library—applied pointwise to each function. The underlying handler introduces a reader monad transformer for the global variables. We assume they have been initialized as part of an initial configuration phase. The local registers are handled into a standard state monad transformer.

3.5 Thread interleaving

The *interleaving* combinator builds a nondeterministic model for a whole multithreaded program from the model of each function, including an initial main function. The jest of this interleaving stage is to interpret away the `ThreadE` events from the local models and build an interleaving semantics. This stage should also retain enough information to allow us to choose a specific memory model in a later stage.

Unfortunately, one easily observe that the generic methodology of defining an handler and relying on `interp` is not applicable to `ThreadE` events. Indeed, `interp` performs a substitution of events by their (single-shot) handler over a single CTree, while our interleaving combinator will necessarily have to take multiple CTrees as inputs, representing the multiple tasks currently running.

We therefore handcraft a new co-recursive combinator `interleave fns fid tasks`. This combinator is parameterized by the list `fns` of models of the functions in scope, and carries recursively two pieces of information as argument: (1) the next fresh thread ID `fid` to be used; and (2) the run-time mapping `tasks` from thread IDs to their (deterministic) models still waiting to be interleaved.

At each co-recursive call, the `interleave` function first checks whether its work is done, i.e., the `tasks` map is empty, otherwise it proceeds to:

1. nondeterministically pick one thread ID `id` to focus on;
2. retrieve the first transition[6] that the focused code can take;
3. if the step is a spawn event, extend the `tasks` map with a fresh thread initialized to the corresponding task, and otherwise take an annotated version of the transition.

[6]We elide details, but point out to the interested reader that retrieving this first step is not completely trivial over CTrees: we reuse the `head` combinator from Chappe et al. [10] to this end.

Step 1 introduces nondeterminism in the computation: as observed in $\mathcal{D}_3$, `SchedC` branches (see Fig. 3) are used to pick a thread ID from the domain of the current `tasks` map. Crucially, these branches are delayed ones, they do not introduce a synchronization point: in $\mathcal{D}_3$, all nodes that are not Br_D are memory events.

Step 3 annotates the non-spawn events it interleaves with the identifier of the thread performing them. This additional information is leveraged by the next step of interpretation that is specific to a memory model. We emphasize that this interleaving combinator is hence independent both from the source language, and from the chosen memory model.

The top-level interleaving operator can finally be defined as `interleave fns 2 [(1, main)]`, i.e., by initializing the `tasks` map to the singleton containing thread ID 1 pointing to the model of the main function.

3.6 Interpretation of inter-thread events

Remains at last to interpret the memory events. As suggested by the signature $\mathcal{D}_4$, we proceed by standard interpretation, via `interp`. Events are handled into a state transformer for a data-structure `PSMem`, introducing additional nondeterministic branching over `PSMemC`.

As any other interpretation handler, the memory event handler must compute the result of memory operations locally, based on the given event and the current memory state. In contrast, a vast and successful body of works on concurrent memory models relies on axiomatic models [1, 17, 20, 34, 48] where acyclicity conditions rule out globally invalid traces. While we could similarly capture a superset of the valid traces and trim the valid subset afterwards, it would likely lead to a complex object to reason about, and essentially negate any possibility of extraction (see Section 4).

Fortunately, operational weak memory models have seen increasing traction over the last decade [16, 24, 33, 45, 50]. These approaches typically define nondeterministic LTSs over extended notions of memory, making them a natural fit for monadic interpreters. As discussed in Section 2.2 we base our model on *Promising Semantics*. More specifically, we work with the promise-free subset of Promising Semantics, as defined in [24].

The semantics of this fragment has remained stable over the different iterations of Promising semantics, except for non-atomic accesses that were only introduced more recently [13, 36]. Noticeably, this later addition is similar but not equivalent to LLVM IR's non-atomics in case of data race. We close this gap by sticking to LLVM IR's non-atomic semantics [9] in our formalization on three main points. First, memory writes do not cause undefined behavior. Second, non-atomic reads return an undefined value if they can read from several messages (i.e., they have more than one valid choice of timestamp). Finally, atomic reads return an undefined value if they can read from several messages, including a non-atomic one.

Our Promising interpretation pass introduces `PSUpdateView` branches (see Listing 3) that correspond to the choice of timestamp when a memory access occurs. The returned timestamps are checked against the Promising state to forbid incorrect outcomes such as overlapping messages. In any case, the interpretation of a memory event introduces a `Step` node, which induces a τ-transition (Figure 3).

A first limitation of our model is its lack of support for unordered accesses (called *plain* accesses in Promising). Plain accesses are not particularly challenging to support, but they add complexity to the model and have a limited use as they do not appear in C-like languages nor in hardware memory models.

Load-store reorderings. A second, more fundamental, limitation of our model is its lack of support for load-store reorderings, as is also the case for e.g. the RC11 model [34], because of their complex semantic implications. Our memory model is thus stronger than full-fledged Promising memory models, and consequently stronger than the LLVM IR memory model. This has several consequences on the applicability of our development. First, it distances us from the actual semantics of LLVM IR: if we prove the validity of an optimization under our semantics, we have no formal guarantee that it is also valid for LLVM IR programs that contain monotonic accesses. Note however that such minor discrepancies are common in efforts of mechanizing semantics of programming languages, especially related to memory models. For instance, the full Promising model is not perfectly faithful to the semantics of C or LLVM IR because of a slightly simplified handling of sequentially consistent access [24].

A second consequence is that a hypothetical verified compiler from our mechanized LLVM IR semantics to an architecture with a weak memory model that exhibits load-store reordering (this includes ARMv8, but not x86) would have to introduce fences around monotonic accesses, so that the compilation does not introduce undesirable additional behaviors. This need for more fences unsurprisingly induces a performance cost on out-of-order architectures. The overhead can be quantified: Ou and Demsky [44] compare the performance overhead of 6 compilation techniques from C/C++ to ARMv8 that forbid load-store reorderings, and measure an average performance overhead ranging between -0.3% and 3.6% depending on the compilation technique chosen. Note however that the correctness of these compilation techniques has not been formally proved, and Lee et al. [36] have pointed out that the suggested technique with the lowest overhead (bogus conditional branch insertion) can be unsound in some contexts.

The most obvious way to support load-store reorderings would be to support promises, a feature from Promising Semantics that we left out. Supporting promises in themselves is not particularly challenging, the problem stems from the undesirable out-of-thin-air behaviors that appear. In Promising Semantics, they are prevented using a sophisticated *certificate* mechanism that cannot be naturally captured by the CTree `interp` combinator. Instead, just like the interleaving pass (Section 3.5), we would need a custom combinator, and because of the very nature of certificates, this combinator would build CTrees that carry proof terms. Sticking to a memory model without promises and certificates makes our development much simpler, and does not limit the range of programs that we can model.

Dependency tracking is another, more operational, option for load-store reordering support, as suggested by Ou and Demsky [44]. It has, to our knowledge, not been explored for Promising-like models. In practice, it would amount to annotate all values with a list of stores they depend on, and checking that no memory store depends on itself. We leave this perspective to future work.

4 Executability

After the final stage, the CTree modeling a program, given initial global and local environments, only contains `Step`, Br_D and `Ret` nodes. It therefore has no unimplemented effect left. Its remaining branches are more precisely `Sched` branches that determine which thread will execute next; memory-model-specific branches such as the choice of timestamp when a memory access occurs in the promising model; `Step` nodes introduced by the interpretation of memory events (Section 3.6); and `Guard` steps introduced all along.

The model can therefore be used for testing, by recursively crawling through the tree. In particular, it suffices to provide an interpretation of the `Sched` and memory-model branches to compute a valid execution of the program. Note that this interpretation can be performed either in Coq, or in OCaml after extraction.

To illustrate the approach on the Coq side, we provide a round-robin scheduler and a pseudo-random scheduler for `Sched` events. For the nodes branching on Promising timestamps, we define two interpretations: one that returns the maximal timestamps, leading to a sequentially consistent execution; and one that chooses a random valid timestamp. Put together with the interpretation stack, this gives us an extracted end-to-end executable interpreter able to simulate an execution of a μ_{IR}^{thread} program.

Thanks to the extraction feature of Coq, we complement these schedulers with a collecting interpreter written in OCaml, that returns all the possible outcomes of a program. As it naïvely explores every possible branch of a CTree, it naturally does not scale, but running it on litmus tests illustrates our model and builds confidence in the correctness of our semantics.

The artifact associated with this paper runs a few litmus tests under several memory models (see Section 5.3) using the collecting interpreter. Its outputs illustrate that the set of outcomes of a program depends on the chosen memory model, and on the chosen ordering annotations.

This executability of the semantics at little additional cost is a key property of definitional monadic interpreters. This had been illustrated already in Vellvm but their interpretation stack eventually splits into a propositional model and an executable interpreter that handle nondeterminism (e.g., undefined behavior and undefined values) differently. Our development goes further in this direction as the CTrees branching nodes provide a unified framework that fully captures nondeterminism while remaining executable.

5 Meta-theory

We sketch three meta-theoretical aspects of our model, laying ground for the future extension of Vellvm with concurrency and memory models. First, we establish that equivalence at each semantic domain is a congruence for its layer of interpretation. When possible, we do so by strengthening the generic meta-theory of CTrees. Second, we establish an operational characterisation of the model at the μ_{IR}^{thread} level. Finally, we introduce alternate memory models and illustrate their use in the modeling pipeline.

5.1 Transporting equivalences through the model

Following a modular design to build our model has benefits in terms of maintainability, extensibility, and code reuse. But as advocated abstractly by Yoon et al. [57], and concretely in Vellvm [58], it also enables us to look at programs under increasingly complex semantic lenses. Consider for example the block fusion optimization proven in [58]: two blocks that are the only successor/predecessor of one another may be fused. While the optimization modifies the control flow of the function, and hence requires a non-trivial coinductive proof, it precisely preserves the trace of occurring events. It can therefore be proven independently from any piece of state. Crucially, this proof can be transported to the full model, because each layer of interpretation preserves the equivalence at the previous semantic layer. We establish similar transport theorems for our model: although the presence of threads complicates greatly the overall semantics of the language, the same proof for block fusion should remain valid!

Figure 6 spells out the precise statements we prove.[7] We work with equivalences built atop of strong bisimilarity of CTrees, written $\sim$, and lift it pointwise to lists and functions.

[7]Note: there is nothing to prove for $\mathcal{I}_{\mathsf{repr}}$, since syntactic equality at the source is preserved trivially.

$$\frac{\forall i,\ t_i \sim u_i}{\forall g\ \bar{l},\ \mathcal{I}_{\mathsf{var}}(\overline{gs}, \bar{t})\ g\ \bar{l} \sim \mathcal{I}_{\mathsf{var}}(\overline{gs}, \bar{u})\ g\ \bar{l}}$$

$$\frac{\forall g\ i\ l,\ (t\ g)_i\ l \sim (u\ g)_i\ l}{\forall g\ \bar{l},\ \mathcal{I}_{\mathsf{thread}}(t)\ g\ \bar{l} \sim \mathcal{I}_{\mathsf{thread}}(u)\ g\ \bar{l}}$$

$$\frac{\forall g\ l,\ t\ g\ l \sim u\ g\ l}{\forall g\ \bar{l}\ m,\ \mathcal{I}_{\mathsf{mem}}(t)\ g\ (m, \bar{l}) \sim \mathcal{I}_{\mathsf{mem}}(u)\ g\ (m, \bar{l})}$$

Figure 6. Equivalence preservation by interpretation

Lists are indicated with an overline, and we access their elements with a subscript.

The proof methodology is fundamentally different for the congruence of $\mathcal{I}_{\mathsf{var}}$ and $\mathcal{I}_{\mathsf{mem}}$ on one hand, and $\mathcal{I}_{\mathsf{thread}}$ on the other. The latter, interleaving the threads, is hand-crafted: its proof of congruence must therefore be handmade as well. The proof is slightly tedious because it involves lists of pointwise bisimilar CTrees, but there is no major difficulty to it—we elude its details. The other two cases however are directly defined in terms of the CTree combinator `interp`. Their congruence can therefore be derived from a generalization of Lemma 5.1 from [10].[8]

More specifically, we say that a CTree is *quasi-pure* if every transition it can take is a value transition (in which case the CTree is actually pure), or if every transition it can take deterministically leads to a Ret leaf. A stateful handler is said to be *quasi-pure* if for all input states, it implements every event into a quasi-pure CTree. Assuming that `h` is quasi-pure, `interp h` is a monad morphism that transports equivalences:

Theorem 5.1. *If* `h : E ⤳stateT S (ctree B F)` *is quasi-pure, then*

$$\forall t\ u,\ t \sim u \implies \texttt{interp}\ h\ t\ s \sim \texttt{interp}\ h\ u\ s.$$

Our handlers for variable accesses and for memory accesses are quasi-pure, thus this theorem implies the first and third cases of Figure 6.

5.2 An operational perspective on the model

While we value the modularity of our construction, our layered view is difficult to relate to a more intuitive and operational view of the semantics of the language. To alleviate this issue, we provide an equational mean to decompose the semantics into syntax-level atomic steps. More precisely, we prove that interleaving partial models is equivalent to picking nondeterministically a live thread identifier, performing the model of its head instruction, and continuing. That is, omitting quantifiers:

```
interleave (ℐvar (ℐrepr fns) g) fid (ℐvar (ℐrepr p) g l)
                                      ~
tid          ←  brD (Sched p);;
(fid',p',l') ←  step fns fid p g l;;
interleave (ℐvar (ℐrepr fns) g) fid' (ℐvar (ℐvar p') g l')
```

Where `step fns fid p g l` is a function that looks up the syntactic code of `fid` in `p`, and depending on the head instruction either computes the result of the terminator, extends the list of threads with a newly created one associated to the corresponding syntactic code in `fns`, or inserts the model of the memory operation terminated with the register update of the instruction.

For this equation to hold up-to strong bisimulation, it is crucial that each source instruction results in a step in the model at the $\mathcal{D}_3$ level: this is the motivation behind the introduction of `Yield` events when representing pure expressions and terminators mentioned in Section 3.3. Indeed, `Yield` events are introduced whenever the interpretation of an instruction would generate no observable step, to enforce that these instructions are observed at the right granularity.

5.3 Models over alternate memory models

As described in Section 3.6, we plug in the model for μ_{IR}^{thread} a weak memory model based on a promise-free Promising Semantics, striking a balance between simplicity and completeness of support for LLVM IR's ordering annotations. Looking ahead, one may need similar models against different memory models: whether it is to prove correct a front-end against a sequentially consistent source language, a back-end against x86's Total Store Ordering (TSO) model [45], or to switch to a simpler model when considering data-race free μ_{IR}^{thread} programs. More generally, any operational memory model can be implemented in our model.

Such applications are far out of the scope of the present paper, focused on the design of our initial infrastructure and meta-theory. Nonetheless, we already illustrate the flexibility of the approach by additionally implementing in our library SC and TSO memory models. We furthermore prove that the SC model of an μ_{IR}^{thread} program , sharing the first three layers of Figure 4, always refines (i.e., is simulated by) its TSO and Promising counterparts.

SC. Recall that the sequential consistency (SC) model is based on the assumption that there is a global memory that is consistent between all threads. The SC state consists of a simple mapping from addresses to values, and all threads share this view of memory. Because the SC model is already as strongly ordered as it can be, fences and memory annotations have no effect.

TSO. In a typical x86 processor, each processor core has its own L1 and L2 caches, and a slower but larger L3 cache is shared between all the cores. In a concurrent setting, a memory write can thus end up in a *local* cache. It will only

[8]This more general result is implemented in the current version of the CTree library.

become visible to the threads of the program that are running on another core when the corresponding cache line is flushed, or when a fence instruction is encountered. TSO takes this caching behavior into account by representing the concurrent memory as a global memory that maps addresses to values (representing the actual RAM and the shared cache levels), plus one local memory per thread (representing the lower cache levels). When a thread writes a value, it goes to its local memory. When it reads a value from memory, it tries to read it from its local memory, and if the address is not known locally it falls back to the global memory.

In this context, a TSO state consists of a mapping from addresses to values that represents the shared memory, plus one such mapping for each thread, that represents the thread-local TSO buffer. As TSO is stronger than the monotonic memory model, only acquire/release and sequentially consistent fences have an effect: they flush the buffer of the relevant thread. In addition to that, the TSO interpretation introduces `TSOFlush` internal branches after each interpreted access to memory. These branching events choose a list of threads whose buffers should be flushed. Thanks to these events, TSO buffers can be emptied at any time.

```
Variant TSOFlushC : Type → Type :=
| TSOFlush (mem: TSOMem) : TSOFlushC (list thread_id)
```

Proofs of simulation. The proofs of simulation between different memory models establishes that SC refines TSO and Promising, meaning that it allows less behaviors. In the following, we focus on the proof that SC refines TSO, but the principle is similar for the Promising proof. We rely on a generic theorem on `interp`:[9] in order to prove that for any CTree with memory events, its TSO interpretation simulates its SC interpretation, it suffices to prove that the TSO interpretation of any memory event, considered in isolation, simulates the SC interpretation of the same memory event.

Theorem 5.2. $\forall$(`h : E` $\rightsquigarrow$ `stateT S (ctree F B)`) (`h' : E` $\rightsquigarrow$ `stateT S' (ctree F' B')`) ($\mathcal{I}$: `rel S S'`) $(\forall e\ s\ s', s\ \mathcal{I}\ s' \implies h\ e\ s \lesssim_{\uparrow(\mathcal{I}\times eq)} h'\ e\ s') \implies \forall t\ s\ s', s\ \mathcal{I}\ s' \implies$ `interp h` $t\ s \lesssim_{\uparrow(\mathcal{I}\times eq)}$ `interp h'` $t\ s'$.

Given two stateful handlers for the same events and an invariant $\mathcal{I}$ to compare these states, this theorem guarantees that if the handlers always produce CTrees in $(\mathcal{I} \times eq)$-simulation (i.e., the "state monad" part of the return value verifies the invariant $\mathcal{I}$ and the rest of the return value is the same on both sides), then the interpretations of any CTree with the two possible handlers will be in $(\mathcal{I} \times eq)$-simulation.

While both the SC and TSO interpretations interpret a CTree into a state monad, their respective states are of different types (`SCMem` and `TSOMem` in our development), hence the need for this heterogeneous version. In our proof, the invariant between SC and TSO states is made of two conditions: the SC memory must be equal to the global TSO memory, and the thread-local TSO memories must all be empty. With this invariant, the proof approach becomes clear: whenever a write happens, we immediately flush the TSO buffer of the relevant thread, so that it is immediately visible to all threads, as with SC.

[9]This theorem does not appear in [10], but is implemented in the current version of the library that we rely on.

6 Related work

Formal semantics of C and LLVM IR.. Although switching from ITrees to CTrees, our work follows closely Zakowski et al.'s Vellvm development [58]. Their work, as ours, put the emphasis in building models allowing for testing, but also suitable for the formal verification of tools and program optimizations.

Many others have proposed formalization of various parts of the C or LLVM IR languages. For C, notable examples include CompCert [38] and its extensions to memory aware programs [5], Krebbers and Wiedijk [31]'s typed C11 semantics, Memarian et al.'s modelling of pointer provenance [43]. Specifically over LLVM IR, Crellvm [26] shares common objectives with Vellvm, while Alive [41, 42], by taking a lighter weight approach, has had impressive results in bug finding through bounded model checking.

All these projects emphasize the importance and difficulty of modelling industrial languages: they however, like Vellvm, restricted themselves to the sequential fragment. By contrast, CompCertTSO [55] has impressively extended CompCert with a TSO model built via a synchronisation machine. They have used their semantics to prove fence elimination optimizations but our approach is more modular in terms of memory model, semantics and concurrency constructs thanks to the theoretical framework of CTrees.

Specifically for LLVM IR, the K-LLVM framework [40], based on the $\mathbb{K}$-framework [52], provides a very complete, executable semantics for LLVM IR with threads. We are however not aware of any formal proof conducted on their semantics. On the contrary, [9] uses event structures to reason on the semantics of acquire/release and non-atomic accesses in LLVM IR, with pen-and-paper compilation proofs from C11 and to hardware models.

Concurrent memory models. An extensive body of works studies concurrent memory models under an axiomatic lens, where allowed behaviors are captured through acyclicity conditions. It has been notably instrumental in clarifying the behavior of modern processors [1, 2].

However, fundamental to our interpretation stack is the ability to define a weak memory model in an operational way. As thoroughly discussed through the paper, we specifically leverage the *Promising Semantics* line of work [12, 13, 24, 36, 37]. While we re-use the base formalism of Promising Semantics, recovering their meta-theory in our formalism is largely left to future work. A possible starting point could be results reducing nondeterminism around non-atomic memory accesses for the verification of thread-local optimizations [59].

Denotational approaches, seeking compositionality, have been developed for concurrent shared-memory-based models over the years. In particular, Brookes' seminal work [7] introduces an elegant denotational trace semantics for sequential consistency. This approach was quickly extended to TSO [22], and later on to weaker memory models using partially ordered multisets (pomsets) [27]. Concurrency in these languages stems from a binary parallel operator, which does not quite fit the kind of C-like imperative language we aim at modelling: our thread scheduling relies on a global view on a list of identified running threads, while a parallel operator leads to more implicit hierarchical scheduling.

It seems that Brookes' work resembles our approach provided we swap the thread interleaving and the inter-thread interpretation passes, and introduce a *stateful* Yield event that sends the memory state to the scheduler and obtains an updated version. Specifying such a commutation and comparing closely the two approaches is left to future work.

Recently, Dvir et al. [14] have proposed a monadic denotational semantics for sequential consistency with a yield operator based on Brookes' traces. This model was later extended to an acquire/release memory model [15], based on the acquire/release fragment of Promising Semantics (a restriction of the promise-free model we use). Beyond the shared context, they rather focus on a pen-and-paper equational theory. Nevertheless, support for the program transformations mentioned in [15] would be an interesting perspective for our Coq development.

Other denotational approaches to weak memory models do not build an interleaving semantics but carry a partial order of dependencies between events. Such approaches rely on event structures [46] or on partially ordered multisets [21, 23, 27].

Bridging the gap with the hardware. While we focus on LLVM IR, a natural perspective would be the verification of an efficient back-end. Faithfully modeling modern hardware is however a major endeavour in itself. IMM [48] is an axiomatic semantics that provides a standard intermediate model bridging the gap between programming language concurrent memory models and axiomatic hardware models; it is meant to factor out proofs of compilation correctness. It notably supports Promising Semantics as a source model.

On a level closer to the architecture, Sail is a DSL for describing formally the behaviour of machine-level instructions [3]. It has been used to give executable operational semantics to the Power [16] and ARMv8 [49] memory models. While we seem to strike for a sensibly different angle at the time, Sail is interesting in that it allows local thread behaviour to be translated into a free monad over an effect datatype. It would seem rather straightforward to interpret this monad into a CTree, and use the framework presented in this paper to reason formally about it, in the presence of concurrent threads.

7 Discussion and future work

The nature of the layered semantics we construct raises a natural question: in which order should the various kinds of events be interpreted? Currently, events that do not involve any communication between threads (i.e., variable accesses) are interpreted before the interleaving pass, and events that involve communication between threads (i.e. memory accesses) are interpreted after.

We have experimented with placing the interpretation pass that deals with variable accesses after the interleaving, but the resulting model turned more difficult to manipulate. In the approach we chose however, the need for extraneous Yield events that the pre-interleaving interpretation of memory events must introduce seem rather contrived.

In order to reconcile these two approaches, a commutation result between thread-local handlers and the interleaving could be established. Such a result would necessitate a transformation from a handler for thread-local events into a post-interleaving handler. Conversely, generating pre-interleaving handlers from post-interleaving handlers would also be possible through the use of a Yield event. These commutation theorems would make clearer the role of the Yield events in the pre-interleaving variable access interpretation pass. We leave all of this to future work.

Future work. In this paper, we have built a semantics for a subset of LLVM IR focused on concurrency. Thanks to the CTree data structure and the accompanying infrastructure, this semantics is built in layers, separating threading, variable access, and memory access concerns. As a result, parts of the model can be reused in other contexts, or tuned piecewise to accommodate alternate semantics. CTrees additionally make our semantics executable, and provide powerful tools to establish meta-theoretical results. However, we have only laid the foundations and are still far away from a full extension of Vellvm to concurrency as CompCertTSO [55] did for CompCert [38].

On a long term basis, our work lays a foundation for the formal verification of optimizations and compilation passes, taking into account a concurrent memory model in a particularly modular way. Building upon the current artifact that formalizes a minimalistic subset of LLVM IR, we first envision to support a more complete language. To reach this goal, we believe it should not raise any major theoretical challenge to integrate our work into Vellvm. Indeed, most of Vellvm's features we omitted (functions, typing) are orthogonal to concurrency. The most complex part would probably be the handling of undefined values and undefined behaviors, but we believe CTrees would provide a better behaved model for them than the current propositional approach.

A Implementation of thread creation

As an intermediate representation, LLVM IR does not specify how threads can be created, this is left to higher-level programming languages and APIs. By itself, our `spawn` syntax is too low-level to be practical, in particular it does not enforce a consistent view on memory between the caller thread and the freshly-created one. However, it is parameterized by thread initialization and cleanup functions whose code is respectively prepended and appended to the code of the actual task to run. We can use these functionalities to implement more realistic thread handling semantics[10]. We base the semantics of thread creation and joining on `thrd_create` and `thrd_join` from the C11 standard library [20]. It is similar to POSIX `pthread_create` and `pthread_join`, but the interactions between these functions and the memory model are more clearly specified in the C standard than in the POSIX one.

Following the C standard, the creation of a thread *synchronizes with* the beginning of the execution of said thread, meaning that memory writes that were visible to the creating thread are made visible to the created thread. Likewise, the end of the execution of a thread synchronizes with the `thrd_join` caller. The semantics of such synchronization corresponds to acquire/release accesses, and can thus be modeled using those at little additional cost: the parent writes the thread argument to memory using a release write, and the child acquire-reads it at the beginning of its execution, which materializes the `synchronizes-with` edge.

In our Coq development, the thread creation and join operations are directly implemented in μ_{IR}^{thread} on top of the low-level spawn instruction. `thrd_create` is a macro (a Coq function that generates μ_{IR}^{thread} code) that accepts two arguments: the function identifier of the thread to spawn, and a value that is passed to it. It returns a pointer to the thread data structure. Then, the macro `thrd_join`, given such a pointer, waits for the completion of the corresponding thread and returns its final result.

References

[1] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2), jul 2021. doi:10.1145/3458926.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. doi:10.1145/2627752.

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71. doi:10.1145/3290384.

[4] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and 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. *Proc. ACM Program. Lang.*, 8(ICFP), aug 2024. doi:10.1145/3674652.

[5] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. Compcerts: A memory-aware verified C compiler using a pointer as integer semantics. *J. Autom. Reason.*, 63(2):369–392, 2019. URL: https://doi.org/10.1007/s10817-018-9496-y, doi:10.1007/S10817-018-9496-Y.

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

[7] Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. URL: https://www.sciencedirect.com/science/article/pii/S0890540196900565, doi:https://doi.org/10.1006/inco.1996.0056.

[8] Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005. doi:10.2168/LMCS-1(2:1)2005.

[9] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an llvm fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 100–110. IEEE Press, 2017.

[10] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571254.

[11] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth handling of nondeterminism. *ACM Trans. Program. Lang. Syst.*, 45(1):5:1–5:43, 2023. doi:10.1145/3579834.

[12] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. Modular data-race-freedom guarantees in the promising semantics. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 867–882, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454082.

[13] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. Sequential reasoning for optimizing compilers under weak memory concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 213–228, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523718.

[14] Yotam Dvir, Ohad Kammar, and Ori Lahav. An algebraic theory for shared-state concurrency. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, pages 3–24, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-21037-2_1.

[15] Yotam Dvir, Ohad Kammar, and Ori Lahav. A denotational approach to release/acquire concurrency. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14577 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2024. doi:10.1007/978-3-031-57267-8_5.

[16] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)*, pages 635–646, December 2015. doi:10.1145/2830772.2830775.

[10] If we supported function calls (as they are in Vellvm), we could use this instead of the prepend/append mechanism but not having functions reduces the overall complexity of the development.

[17] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. An axiomatic basis for computer programming on the relaxed arm-a architecture: The axsl logic. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi:10.1145/3632863.

[18] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 317–331, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[19] Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. A type system for extracting functional specifications from memory-safe imperative programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi:10.1145/3485512.

[20] ISO/IEC. *ISO/IEC 9899:2011*. 2011.

[21] Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA):194:1–194:30, 2020. doi:10.1145/3428262.

[22] Radha Jagadeesan, Gustavo Petri, and James Riely. Brookes is relaxed, almost! In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'12, pages 180–194, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-28729-9_12.

[23] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi:10.1145/3498716.

[24] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017. doi:10.1145/3009837.3009850.

[25] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 326–335. ACM, 2015. doi:10.1145/2737924.2738005.

[26] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified credible compilation for llvm. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 631–645, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192377.

[27] Ryan Kavanagh and Stephen Brookes. A denotational semantics for sparc tso. *Electronic Notes in Theoretical Computer Science*, 336:223–239, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). URL: https://www.sciencedirect.com/science/article/pii/S1571066118300288, doi:https://doi.org/10.1016/j.entcs.2018.03.025.

[28] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. URL: http://doi.acm.org/10.1145/2804302.2804319, doi:10.1145/2804302.2804319.

[29] Jérémie Koenig and Zhong Shao. Compcerto: compiling certified open C components. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1095–1109. ACM, 2021. doi:10.1145/3453483.3454097.

[30] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 234–248. ACM, 2019. doi:10.1145/3293880.3294106.

[31] Robbert Krebbers and Freek Wiedijk. A typed C11 semantics for interactive theorem proving. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 15–27. ACM, 2015. doi:10.1145/2676724.2693571.

[32] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014. URL: https://cakeml.org/popl14.pdf, doi:10.1145/2535838.2535841.

[33] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 649–662, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837643.

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062352.

[35] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[36] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. Putting weak memory in order via a promising intermediate representation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi:10.1145/3591297.

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

[38] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. URL: https://hal.inria.fr/inria-00415861, doi:10.1145/1538788.1538814.

[39] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–31, 2022. doi:10.1145/3527324.

[40] Liyi Li and Elsa L. Gunter. K-LLVM: A Relatively Complete Semantics of LLVM IR. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.7, doi:10.4230/LIPIcs.ECOOP.2020.7.

[41] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for llvm. PLDI '21, 2021. doi:10.1145/3453483.3454030.

[42] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. PLDI 15,

pages 22–32. ACM, 2015. doi:10.1145/2813885.2737965.

[43] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019. doi:10.1145/3290380.

[44] Peizhao Ou and Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi:10.1145/3276506.

[45] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[46] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. Modular relaxed dependencies in weak memory concurrency. In Peter Müller, editor, *Programming Languages and Systems*, pages 599–625, Cham, 2020. Springer International Publishing.

[47] Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In Bart Jacobs, Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014*, volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 273–288. Elsevier, 2014. URL: https://doi.org/10.1016/j.entcs.2014.10.015, doi:10.1016/J.ENTCS.2014.10.015.

[48] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290382.

[49] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi:10.1145/3158107.

[50] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-arm/risc-v: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314624.

[51] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[52] Grigore Roşu and Traian Florin Şerbănută. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. Membrane computing and programming. URL: http://www.sciencedirect.com/science/article/pii/S1567832610000160, doi:https://doi.org/10.1016/j.jlap.2010.03.012.

[53] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for noninterference with interaction trees. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 29:1–29:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.29, doi:10.4230/LIPICS.ECOOP.2023.29.

[54] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. Conditional contextual refinement. *Proc. ACM Program. Lang.*, 7(POPL):1121–1151, 2023. doi:10.1145/3571232.

[55] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), jun 2013. doi:10.1145/2487241.2487248.

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

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

[58] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi:10.1145/3473572.

[59] Junpeng Zha, Hongjin Liang, and Xinyu Feng. Verifying optimizations of concurrent programs in the promising semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 903–917, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523734.

[60] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ITP.2021.32, doi:10.4230/LIPICS.ITP.2021.32.

[61] Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. Fully composable and adequate verified compilation with direct refinements between open modules. *Proc. ACM Program. Lang.*, 8(POPL):2160–2190, 2024. doi:10.1145/3632914.

[62] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103621.2103709.

[63] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2013. doi:10.1145/2499370.2462164.