



HAL
open science

A concurrency model based on monadic interpreters: executable semantics for a concurrent subset of LLVM IR

Nicolas Chappe, Ludovic Henrio, Yannick Zakowski

► **To cite this version:**

Nicolas Chappe, Ludovic Henrio, Yannick Zakowski. A concurrency model based on monadic interpreters: executable semantics for a concurrent subset of LLVM IR. 2024. hal-04594073

HAL Id: hal-04594073

<https://hal.science/hal-04594073v1>

Preprint submitted on 30 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A concurrency model based on monadic interpreters (draft)

Executable semantics for a concurrent subset of LLVM IR

Nicolas Chappe ✉ 

Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France

Ludovic Henrio ✉ 

Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France

Yannick Zakowski ✉ 

Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP, F-69342, 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 (CTrees) library provides generic tools to craft such monadic interpreters while supporting concurrency with nodes encoding non-deterministic choice. This monadic approach allows the definition of programming language semantics that is *modular*, *compositional* and *executable*.

This paper demonstrates the use of CTrees to formalize semantics for concurrency and weak memory models in Coq. Our semantics is built in successive stages, interpreting each aspect of the semantics separately. We instantiate the approach by defining the semantics of a minimal concurrent subset of LLVM IR with a memory model based on Kang et al’s work on *Promising Semantics*, but 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 new results on the notions of (bi)similarity of CTrees, we establish the equational theory of our constructions, and show how to transport equivalences through our layered construction. Finally, our model is executable, hence we can test the semantics by extraction to OCaml.

2012 ACM Subject Classification Author: Please fill in 1 or more `\ccsdesc macro`

Keywords and phrases semantics, concurrency, Coq, LLVM

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In recent years, large-scale verification of industrial-strength software has become increasingly common [52] 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, 56, 63].

In the Coq ecosystem, the *Interaction Trees* (ITree) library by Xia et al. [58, 59] has been influential over the recent years as a rich semantic toolbox for modelling 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, 62], transactional objects [39], non-interference [55], or memory-safe imperative programs [19].



© Author: Please provide a copyright holder;

licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

52 While the Vellvm project takes its roots over a decade ago [64, 65], Zakowski et al. have
 53 restarted the project on denotational foundations using the ITree library [60]. The approach
 54 has been celebrated by Zakowski et al. through the mantra “*a compositional, modular, and*
 55 *executable semantics*”. Compositional in that it is built by structural recursion on the syntax,
 56 and defines the meaning of open programs. Modular in that it defines and compose the
 57 semantics of each effect as independent handlers. Executable in that the model allows for
 58 the extraction of a verified executable interpreter suitable for testing.

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

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

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

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

82 We also demonstrate how the modularity of the approach enables flexible reuse of the
 83 interpretation passes. Our results are formalized in the Coq proof assistant, and provided as
 84 an open source artifact.¹

¹ <https://github.com/micro-vellvm-concurrency/micro-vellvm-concurrency>

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 ARM do not ensure *sequential consistency* (SC) (i.e., writes to memory are immediately visible to all threads).

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 targetting hardware with a weaker memory model in order 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 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 information².

- 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. It is slightly stronger than unordered accesses. 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 (for instance, an *acquire read* reads a value that comes 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.2) may return undef as it can read both the initial 0 or the 2 from (A.1). By contrast, the monotonic load 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.

² <https://llvm.org/docs/LangRef.html>

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

XX:4 A concurrency model based on monadic interpreters

```
thread A                                thread B
1 store monotonic 2, @x                  1 %1 = load @x
2 fence release                          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).

```
thread A                                thread B
1 store monotonic 2, @x ; t=2            1 store monotonic 2, @y ; t=2
2 store monotonic 1, @y ; t=1            2 store monotonic 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). The comments indicate a possible assignment of timestamps at which load and store operations occur.

127 2.2 Promising Semantics

128 We seek a formal memory model that supports the different LLVM IR memory access
129 operations (read, write, read-modify-write and fence) and orderings. We furthermore need the
130 model to be *operational*: by defining locally the next available transitions of the system, such
131 models fit better in the CTree formalism. Promising Semantics [24] is one such operational
132 weak memory model, and has been extensively studied over the past few years [37, 13, 36, 61].

133 In its most basic form, Promising semantics uses two components to model a shared
134 memory: a global set of *messages* and per-thread *views*. The set of *messages* materializes
135 the past writes to memory. A message mainly contains an address, a value, and a timestamp.
136 The timestamps have a per-address semantics in the sense that each memory address has its
137 own totally ordered timeline of its past stores.

138 The global Promising state also contains thread states. Each thread has a *view* that
139 remembers for each address the timestamp of its last performed operation. The timestamps
140 in the view of a given thread can only increase over time, but not necessarily to the maximal
141 possible timestamp. We omit here details about additional views stored in global state,
142 thread states and messages, used for sequentially consistent and acquire/release accesses.
143 The example in Listing 2, adapted from [24], demonstrates how timestamps enable store-store
144 reordering in Promising semantics. *a* and *b* can both be assigned 1 in the same execution.

145 We stress that this short description of promising semantics is simplified. The full-fledged
146 promising semantics supports two other important features. First, timestamps are actually
147 intervals of the form (from, to], this allows modelling read-modify-write operations. Second,
148 promising semantics also supports load-store reorderings thanks to *promises*. At any point
149 of an execution, a thread can promise that it will later write some value to some address at
150 some timestamp. Other threads accessing this address can read from this promise as if the
151 future write had already happened. Finally, every promise has to be eventually fulfilled.

152 We only support the first of these two features in our implementation, as described in
153 Section 3.6. This allows us to support all the orderings of an acquire/release semantics, but
154 not the load-store reorderings allowed in the monotonic ordering.



Figure 1 The four kinds of nodes in values of type `ctree E B X`. `r` value of type `X`, `e` external event taken from `E`, and `b` external or internal branching taken from `B`. The edge labels hint at the LTS the structure represents.

2.3 Choice Trees

CTrees, introduced in [10], is a Coq library providing a coinductive [49] data-structure `ctree E B X` of potentially infinite trees. As illustrated in Figure 1, values of this type exhibit four kinds of nodes: leaves carrying values of type `X`, external events (`Vis`) taken from the signature `E`, and two variants of nondeterministic branching (`BrD` and `BrS`) taken from the signature `B`⁴. A signature is a family of types: for instance, an event `e : E nat` indicates the effect expects back a natural number, and hence that the node in the tree branches over `nat`.

While external events encode observable computation transitions, non-deterministic branches have no visible behaviour, 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 1), while `BrD` nodes are truly invisible, capturing a proper internal non-deterministic transition. 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.

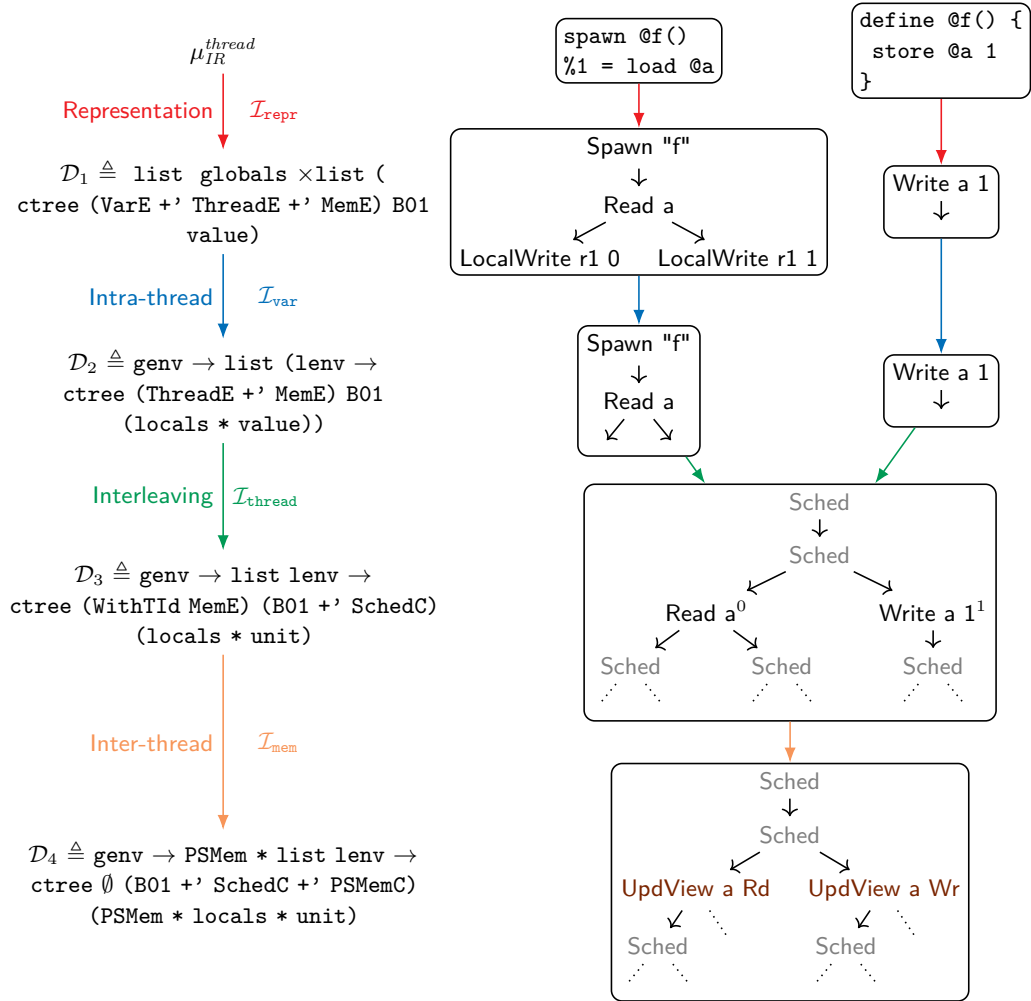
CTrees come with similar combinators as ITrees. It forms a monad, with the traditional `ret` and `bind` constructs. It supports iteration, via the combinator `iter f i` that iterates a loop body `f`, starting from `i`, until an exit signal is reached. Crucial to the construction of models based on these libraries, CTrees also support an `interp h` primitive, that captures the structure is *free* in the parameter `E`. Given a *handler* `h`, that is an implementation of the external events from `E` into an appropriate monad `M`, the function `interp h t` recursively applies `h` over the tree `t`. The resulting monadic computation therefore corresponds to the initial tree, where external events are now implemented internally.

Equivalence of CTrees is implemented as strong bisimilarity over the labelled transition system (LTS) sketched in Figure 1. There is no label on `BrD` nodes: they are not visible in the resulting LTS. The strong bisimulation game hence treats `BrD` nodes in a manner reminiscent of weak bisimulation, albeit subtly different: we refer the interested reader to [10].

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. Note however that beyond this concrete application, the principles and the tools we develop are applicable to other concurrent languages.

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



■ **Figure 2** 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.

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

190 3.1 A semantic model built as an interpretation stack

191 Figure 2 illustrates the construction of the semantic model. It is structured into successive
 192 stages of interpretation, from a source language (μ_{IR}^{thread} in our case study, introduced
 193 hereafter) all the way down to our semantic domain \mathcal{D}_4 , a monadic computation combining
 194 a read-only map of globals, stateful local and memory states, and internalizing the potential
 195 divergence and non-determinism into a CTree. The stack follows four stages:

- 196 ■ *CTree representation.* From the source language, each function is represented into a
 197 (deterministic) CTree. This stage produces a list of CTrees.
- 198 ■ *Intra-thread interpretation.* This stage gives a semantic to thread-local events: this
 199 process can be done point-wise over \mathcal{D}_1 , it is unrelated to the concurrent nature of

```

atom ::= @id | %id | int | bool | undef
exp  ::= atom | atom op atom
aop  ::= atomic_exchange | atomic_add
ord  ::= not_atomic | monotonic | acquire | release | acq_rel | sc
instr ::= exp | alloca (exp) | loadord (exp) | storeord (exp, exp)
      | rmword (aop, exp, exp) | cmpxchgord (exp, exp, exp)
      | fenceord | spawn (fid, fid, fid, x)
term ::= branch (exp, bid, bid) | jmp (bid) | return (exp)
sblock ::= {entry : bid; code : list (fid, instr); term : term}
cfg     ::= {name : fid; entry : id; body : list sblock}
prog    ::= {main : cfg; funs : list cfg; globs : list @id}

```

■ **Figure 3** Syntax for μ_{IR}^{thread} , a minimal subset of LLVM IR

200 the computation. For μ_{IR}^{thread} , this phase deals with accesses to globals and registers,
 201 introducing a read and a state monad transformers in \mathcal{D}_2 .

- 202 ■ *Interleaving.* This pass takes the (deterministic) CTrees modelling the (spawnable)
 203 functions in \mathcal{D}_2 , and builds a singular (nondeterministic) CTree that represents the
 204 concurrent execution of the program. Spawn events are given a semantics at this stage.
- 205 ■ *Inter-thread interpretation.* This last stage gives a semantics to the remaining events, the
 206 ones that are not thread-local; in particular it interprets shared memory accesses. In our
 207 case-study, we build an operational Promising-like memory model supporting non-atomic,
 208 acquire/release, and (partly) monotonic accesses.

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

213 3.2 The source language: μ_{IR}^{thread}

214 Figure 3 depicts the syntax of μ_{IR}^{thread} , our source language. A program includes an identified
 215 *main* function, a list of global variable (*@id*) declarations, and a list of functions ready to
 216 be spawned. These functions take exactly one argument. They are defined as control flow
 217 graphs, i.e., a name, an entry block, and a list of blocks. Blocks contain an identifier, straight
 218 line three address code, and a terminator either returning, or jumping to a new block.

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

227 We leverage this spawn primitive to implement in μ_{IR}^{thread} thread creation and joining,
 228 based on **thrd_create** and **thrd_join** from the C11 standard library [20]. Their semantics,
 229 and our implementation, relies on acquire/release accesses for synchronization. Due to the
 230 absence of function calls in μ_{IR}^{thread} , we use Coq-level macros to generate the code, but would
 231 use source-level functions in a language like Vellvm. Appendix A provides additional details.


```

(* 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: ordering) (k: addr) (v: value) : MemE unit
| ReadWrite (o: ordering) (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

(* Additional 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 : PSMem → thread_id → addr → PSAccess → PSMemC (date * date)

```

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

232 As any production level language, LLVM IR accumulates numerous orthogonal features,
 233 leading to active research even when restricted to its sequential memory model [25, 4]. In
 234 order to keep the complexity of our development reasonable, many LLVM IR features, mostly
 235 unrelated to concurrency concerns (typing, function calls other than via `spawn`, undefined
 236 behaviors, etc.) are not supported in our development. These excluded features are however
 237 supported in Vellvm [60]. We expect that a future integration of our contributions to Vellvm
 238 would only require minor modifications to the way we handle concurrency and memory.

239 3.3 CTree representation for μ_{IR}^{thread}

240 This first step translates the syntax into the semantic domain \mathcal{D}_1 : each function is denoted
 241 into a CTree, and collected into a list, along with the global variables. This process is rather
 242 standard, following closely Vellvm to resolve the control flow, albeit using CTrees rather
 243 than ITrees. In particular, graphs are denoted as a tail recursive fixpoint of the function
 244 mapping block identifiers to their denotation. We refer to Zakowski et al. [60] for details.

245 Crucial to this denotation is the identification of the effects of the language, captured
 246 for now into abstract events. We inventory them in Listing 3: interactions with the local
 247 and global variables (`VarE`), interactions with the shared memory (`MemE`), and multi-threading
 248 events (`ThreadE`). Note that these events only specify a signature at this stage: their semantics
 249 will be refined in the subsequent stages of interpretation; this leaves us, in particular, all
 250 flexibility in choosing the memory model later on.

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

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

260 CTrees are not only parameterized by their interface of events, but also by their interface
 261 of internal branching, and of course by a return type. In \mathcal{D}_1 , the internal branching is
 262 restricted to `B01`, i.e., unary nodes. Consequently, each function is modelled as a CTree with
 263 a single *deterministic* trace. The return type in \mathcal{D}_1 corresponds to the type of dynamic
 264 values. In μ_{IR}^{thread} , dynamic values are restricted to unbounded signed integers that also serve
 265 as pointers, once again to limit the features covered by our language.

266 3.4 Interpretation of intra-thread events

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

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

276 3.5 Thread interleaving

277 The *interleaving* combinator builds a non-deterministic model for a whole multi-threaded
 278 program from the deterministic model of each function, including an initial main function.

279 The jest of this interleaving stage is to interpret away the `ThreadE` events from the local
 280 models and build an interleaving semantics. This stage should also retain enough information
 281 to allow us to choose a specific memory model in a later stage. This transformation is
 282 however too global to be definable via `interp`. We therefore handcraft a new co-recursive
 283 combinator `interleave fns fid tasks`.

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

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

- 290 1. non-deterministically pick one thread ID `id` to focus on;
- 291 2. retrieve the first transition⁵ that the focused code can take;
- 292 3. if the step is a spawn event, extend the `tasks` map with a fresh thread initialized to the
 293 corresponding task, and otherwise take an annotated version of the transition.

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

⁵ 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.

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

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

304 3.6 Interpretation of inter-thread events

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

308 We may already observe that the approach entails a limitation: the valid values resulting
 309 from a read must be captured locally. In contrast, a vast and successful body of works on
 310 concurrent memory models relies on axiomatic models [20, 34, 48, 1, 17] where acyclicity
 311 conditions rule out globally invalid traces. While we could similarly capture a superset of the
 312 valid traces and trim the valid subset afterwards, it would likely lead to a complex object to
 313 reason about, and essentially negate any possibility of extraction (see Section 4).

314 Fortunately, operational weak memory models have seen increasing traction over the
 315 last decade [45, 16, 33, 24, 51]. These approaches typically define non-deterministic LTSs
 316 over extended notions of memory, making them a natural fit for monadic interpreters. As
 317 discussed in Section 2.2 we base our model on *Promising Semantics*. More specifically, we
 318 work with the promise-free subset of Promising Semantics, as defined in [24].

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

328 Our Promising interpretation pass introduces `PSUpdateView` branches (see Listing 3)
 329 that correspond to the choice of timestamp when a memory access occurs. The returned
 330 timestamps are checked against the Promising state to forbid incorrect outcomes such as
 331 overlapping messages. In any case, the interpretation of a memory event introduces a `Step`
 332 node, which induces a τ -transition (Figure 1).

333 Without support for load-store reorderings, our memory model is stronger than full-fledged
 334 Promising memory models, which has a performance cost for the compilation of monotonic
 335 memory accesses [44]. However, supporting promises would be particularly challenging, as
 336 it involves a sophisticated *certificate* mechanism that cannot be naturally captured by the
 337 CTree `interp` combinator.

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

342 4 Executability

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

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

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

359 Alternatively, we implement a collecting interpreter that returns all the possible outcomes
 360 of a program. This interpreter is naively extracted as an OCaml executable, which naturally
 361 does not scale, but running it on litmus tests illustrates our model and builds confidence in
 362 the correctness of our semantics.

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

369 5 Meta-theory

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

376 5.1 Transporting equivalences through the model

377 Following a modular design to build our model has benefits in terms of maintainability,
 378 extensibility, and code reuse. But as advocated abstractly by Yoon et al. [59], and concretely
 379 in Vellvm [60], it also enables us to look at programs under increasingly complex semantic
 380 lenses. Consider for example the block fusion optimisation proven in [60]: two blocks that are
 381 the only successor/predecessor of one another may be fused. While the optimization modifies
 382 the control flow of the function, and hence requires a non-trivial coinductive proof, it precisely
 383 preserves the trace of occurring events. It can therefore be proven independently from any
 384 piece of state. Crucially, this proof can be transported to the full model, because each layer of
 385 interpretation preserves the equivalence at the previous semantic layer. We establish similar

$$\frac{\forall i, t_i \sim u_i}{\forall g \bar{l}, \mathcal{I}_{\text{var}}(\bar{g}\bar{s}, \bar{t}) g \bar{l} \sim \mathcal{I}_{\text{var}}(\bar{g}\bar{s}, \bar{u}) g \bar{l}} \quad \frac{\forall g i l, (t g)_i l \sim (u g)_i l}{\forall g \bar{l}, \mathcal{I}_{\text{thread}}(t) g \bar{l} \sim \mathcal{I}_{\text{thread}}(u) g \bar{l}}$$

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

■ **Figure 4** Equivalence preservation by interpretation

386 transport theorems for our model: although the presence of threads complicates greatly the
 387 overall semantics of the language, the same proof for block fusion should remain valid!

388 Figure 4 spells out the precise statements we prove.⁶ We work with equivalences built
 389 atop of strong bisimilarity of CTrees, written \sim , and lift it point-wise to lists and functions.
 390 Lists are indicated with an overline, and we access their elements with a subscript.

391 The proof methodology is fundamentally different for the congruence of \mathcal{I}_{var} and \mathcal{I}_{mem} on
 392 one hand, and $\mathcal{I}_{\text{thread}}$ on the other. The latter, interleaving the threads, is hand-crafted: its
 393 proof of congruence must therefore be handmade as well—we elude its details. The first two
 394 cases however are directly defined in terms of the CTree combinator `interp`. Their congruence
 395 can therefore be derived from an extension of generic results introduced in [10].

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

401 ► **Theorem 1.** *Quasi-pure stateful handlers. If $h : E \rightsquigarrow \text{stateT } S \text{ (ctree } B \text{ } F)$ is quasi-*
 402 *pure, then $\forall t u, t \sim u \implies \text{interp } h \text{ } t \text{ } s \sim \text{interp } h \text{ } u \text{ } s$.*

403 Proving this theorem using the original bisimilarity for CTrees [10] would be too difficult.
 404 Instead we introduce an alternate, equivalent, definition of strong bisimilarity for CTrees.
 405 Its formal description is out of the scope of this paper, we only provide its intuition and
 406 refer the interested reader to our formal development. While the original CTree simulation
 407 works over the LTS in which BrD nodes are collapsed before hand, we consider these nodes
 408 as so-called ϵ -transitions, treated weakly in our simulation. Building the bisimilarity on top
 409 of this simulation requires more care, it cannot be defined simply using the intersection of
 410 two such simulation half-games. Rather, the simulation keeps track of whether it is in the
 411 left or right bisimulation half-game. The approach is similar in spirit, but not comparable,
 412 to the notion of coupled simulation [54].

413 5.2 An operational perspective on the model

414 While we value the modularity of our construction, our layered view is difficult to relate to
 415 a more intuitive and operational view of the semantics of the language. To alleviate this
 416 issue, we provide an equational mean to decompose the semantics into syntax-level atomic
 417 steps. More precisely, we prove that interleaving partial models is equivalent to picking

⁶ Note: there is nothing to prove for $\mathcal{I}_{\text{repr}}$, since syntactic equality at the source is preserved trivially.

418 non-deterministically a live thread identifier, performing the model of its head instruction,
 419 and continuing. That is, omitting quantifiers:

```
420 interleave ( $\mathcal{I}_{\text{var}}$  ( $\mathcal{I}_{\text{repr}}$  fns) g) fid ( $\mathcal{I}_{\text{var}}$  ( $\mathcal{I}_{\text{repr}}$  p) g l)  

  421 ~  

  422 tid ← brD (Sched p);;  

  423 (fid',p',l') ← step fns fid p g l;  

  424 interleave ( $\mathcal{I}_{\text{var}}$  ( $\mathcal{I}_{\text{repr}}$  fns) g) fid' ( $\mathcal{I}_{\text{var}}$  ( $\mathcal{I}_{\text{var}}$  p') g l')  

  425
```

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

431 For this equation to hold up-to strong bisimulation, it is crucial that each source instruction
 432 results in a step in the model at the \mathcal{D}_3 level: this is the motivation behind the introduction
 433 of `Yield` events when representing pure expressions and terminators mentioned in Section 3.3.

434 5.3 Models over alternate memory models

435 As described in Section 3.6, we plug in the model for μ_{IR}^{thread} a weak memory model based
 436 on a promise-free Promising Semantics, striking a balance between simplicity and richness
 437 of support for LLVM IR's ordering annotations. Looking ahead, one may need similar
 438 models against different memory models: whether it is to prove correct a front-end against
 439 a sequentially consistent source language, a back-end against x86's TSO model [45], or to
 440 switch to a simpler model when considering data-race free μ_{IR}^{thread} 's programs.

441 Such applications are far out of the scope of the present paper, focused on the presentation
 442 of the initial infrastructure. Nonetheless, we already illustrate the flexibility of the approach
 443 by additionally implementing in our library SC and TSO memory models. We furthermore
 444 prove that the SC model of an μ_{IR}^{thread} program, sharing the first three layers of Figure 2,
 445 always simulates its TSO and Promising models.

446 6 Related work and discussion

447 **Formal semantics of C and LLVM IR.** Although switching from ITrees to CTrees, our
 448 work follows closely Zakowski et al.'s Vellvm development [60]. Their work, as ours, put the
 449 emphasis in building models allowing for testing, but also suitable for the formal verification
 450 of tools and program optimizations.

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

457 All these projects emphasize the importance and difficulty of modelling industrial
 458 languages: they however, like Vellvm, restricted themselves to the sequential fragment.
 459 By contrast, CompCertTSO [57] has impressively extended CompCert with a TSO model
 460 built via a synchronisation machine. They have used their semantics to prove fence elimination
 461 optimizations. Specifically for LLVM IR, the K-LLVM framework [40], based on the \mathbb{K} -
 462 framework [53], provides a very complete, executable semantics for LLVM IR with threads.
 463 We are however not aware of any formal proof conducted on their semantics. On the contrary,
 464 [9] uses event structures to reason on the semantics of acquire/release and non-atomic accesses
 465 in LLVM IR, with pen-and-paper compilation proofs from C11 and to hardware models.

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

469 However, fundamental to our interpretation stack is the ability to define a weak memory
470 model in an operational way. As thoroughly discussed through the paper, we specifically
471 leverage the *Promising Semantics* line of work [24, 37, 12, 13, 36]. While we re-use the base
472 formalism of Promising Semantics, recovering their meta-theory in our formalism is largely
473 left to future work. A possible starting point could be results reducing non-determinism
474 around non-atomic memory accesses for the verification of thread-local optimizations [61].

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

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

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

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

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

502 On a level closer to the architecture, Sail is a DSL for describing formally the behaviour
503 of machine-level instructions [3]. It has been used to give executable operational semantics
504 to the Power [16] and ARM [50] memory models. While we seem to strike for a sensibly
505 different angle at the time, Sail is interesting in that it allows local thread behaviour to be
506 translated into a free monad over an effect datatype. It would seem rather straightforward to
507 interpret this monad into a CTree, and use the framework presented in this paper to reason
508 formally about it, in the presence of concurrent threads.

509 Future work

510 The work presented in this paper paves the way towards an extension of Vellvm with
511 concurrency. While the current artifact only formalizes a minimalistic subset of LLVM IR,

512 and although its integration into Vellvm will require significant efforts, we believe it should
513 not face any major theoretical challenge. On a long term basis, our work should thus allow
514 for the formal verification of optimizations and compilation passes, taking into account a
515 concurrent memory model in a particularly modular way. For that purpose, it should be
516 possible in particular to recover in our formalism many meta-theoretical results from the
517 Promising Semantics line of research. We have illustrated in this paper the foundations that
518 give us confidence this objective can be achieved.

519 — References —

- 520 1 Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget.
521 Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2),
522 jul 2021. doi:10.1145/3458926.
- 523 2 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation,
524 testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74,
525 2014. doi:10.1145/2627752.
- 526 3 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray,
527 Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked
528 Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V,
529 and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of*
530 *Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.
531 doi:10.1145/3290384.
- 532 4 Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. A two-phase
533 infinite/finite low-level memory model, 2024. arXiv:2404.16143.
- 534 5 Frédéric Besson, Sandrine Blazy, and Pierre Wilke. Compcerts: A memory-aware verified C
535 compiler using a pointer as integer semantics. *J. Autom. Reason.*, 63(2):369–392, 2019. URL:
536 <https://doi.org/10.1007/s10817-018-9496-y>, doi:10.1007/S10817-018-9496-Y.
- 537 6 Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. Skeletal semantics
538 and their interpretations. *Proc. ACM Program. Lang.*, 3(POPL):44:1–44:31, 2019. doi:
539 10.1145/3290357.
- 540 7 Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information*
541 *and Computation*, 127(2):145–163, 1996. URL: [https://www.sciencedirect.com/science/](https://www.sciencedirect.com/science/article/pii/S0890540196900565)
542 [article/pii/S0890540196900565](https://www.sciencedirect.com/science/article/pii/S0890540196900565), doi:10.1006/inco.1996.0056.
- 543 8 Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2),
544 2005. doi:10.2168/LMCS-1(2:1)2005.
- 545 9 Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an llvm
546 fragment. In *Proceedings of the 2017 International Symposium on Code Generation and*
547 *Optimization*, CGO '17, pages 100–110. IEEE Press, 2017.
- 548 10 Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice
549 trees: Representing nondeterministic, recursive, and impure programs in coq. *Proc. ACM*
550 *Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571254.
- 551 11 Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics:
552 Smooth handling of nondeterminism. *ACM Trans. Program. Lang. Syst.*, 45(1):5:1–5:43, 2023.
553 doi:10.1145/3579834.
- 554 12 Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. Modular data-race-freedom
555 guarantees in the promising semantics. In *Proceedings of the 42nd ACM SIGPLAN International*
556 *Conference on Programming Language Design and Implementation*, PLDI 2021, pages 867–882,
557 New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.
558 3454082.
- 559 13 Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. Sequential reasoning
560 for optimizing compilers under weak memory concurrency. In *Proceedings of the 43rd ACM*
561 *SIGPLAN International Conference on Programming Language Design and Implementation*,

- 562 PLDI 2022, pages 213–228, New York, NY, USA, 2022. Association for Computing Machinery.
563 doi:10.1145/3519939.3523718.
- 564 14 Yotam Dvir, Ohad Kammar, and Ori Lahav. An algebraic theory for shared-state concurrency.
565 In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New*
566 *Zealand, December 5, 2022, Proceedings*, pages 3–24, Berlin, Heidelberg, 2022. Springer-Verlag.
567 doi:10.1007/978-3-031-21037-2_1.
- 568 15 Yotam Dvir, Ohad Kammar, and Ori Lahav. A denotational approach to release/acquire
569 concurrency. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd*
570 *European Symposium on Programming, ESOP 2024, Held as Part of the European Joint*
571 *Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg,*
572 *April 6-11, 2024, Proceedings, Part II*, volume 14577 of *Lecture Notes in Computer Science*,
573 pages 121–149. Springer, 2024. doi:10.1007/978-3-031-57267-8_5.
- 574 16 Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar,
575 and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition,
576 and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International*
577 *Symposium on Microarchitecture (Waikiki)*, pages 635–646, December 2015. doi:10.1145/
578 2830772.2830775.
- 579 17 Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean
580 Pichon-Pharabod. An axiomatic basis for computer programming on the relaxed arm-a
581 architecture: The axsl logic. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi:10.1145/
582 3632863.
- 583 18 Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter G.
584 Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 317–331, Berlin,
585 Heidelberg, 2000. Springer Berlin Heidelberg.
- 586 19 Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer,
587 Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. A
588 type system for extracting functional specifications from memory-safe imperative programs.
589 *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi:10.1145/3485512.
- 590 20 ISO/IEC. *ISO/IEC 9899:2011*. 2011.
- 591 21 Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple
592 model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA):194:1–194:30, 2020.
593 doi:10.1145/3428262.
- 594 22 Radha Jagadeesan, Gustavo Petri, and James Riely. Brookes is relaxed, almost! In *Proceedings*
595 *of the 15th International Conference on Foundations of Software Science and Computational*
596 *Structures, FOSSACS’12*, pages 180–194, Berlin, Heidelberg, 2012. Springer-Verlag. doi:
597 10.1007/978-3-642-28729-9_12.
- 598 23 Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev.
599 The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency.
600 *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi:10.1145/3498716.
- 601 24 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising
602 semantics for relaxed-memory concurrency. *SIGPLAN Not.*, 52(1):175–189, January 2017.
603 doi:10.1145/3093333.3009850.
- 604 25 Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and
605 Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In David Grove
606 and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on*
607 *Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*,
608 pages 326–335. ACM, 2015. doi:10.1145/2737924.2738005.
- 609 26 Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park,
610 Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and
611 Kwangkeun Yi. Crellvm: Verified credible compilation for llvm. In *Proceedings of the 39th*
612 *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: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/pop14.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.

- 665 39 Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C.
666 Pierce, and Steve Zdancewic. C4: verified transactional objects. *Proc. ACM Program. Lang.*,
667 6(OOPSLA1):1–31, 2022. doi:10.1145/3527324.
- 668 40 Liyi Li and Elsa L. Gunter. K-LLVM: A Relatively Complete Semantics of LLVM IR. In
669 Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented*
670 *Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics*
671 *(LIPIcs)*, pages 7:1–7:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für
672 Informatik. URL: <https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.7>, doi:10.4230/LIPIcs.ECOOP.2020.7.
- 674 41 Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2:
675 Bounded translation validation for llvm. PLDI '21, 2021. doi:10.1145/3453483.3454030.
- 676 42 Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct
677 peephole optimizations with alive. PLDI 15, pages 22–32. ACM, 2015. doi:10.1145/2813885.
678 2737965.
- 679 43 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson,
680 Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proc.*
681 *ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019. doi:10.1145/3290380.
- 682 44 Peizhao Ou and Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air
683 results. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi:10.1145/3276506.
- 684 45 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In
685 Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem*
686 *Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer Berlin
687 Heidelberg.
- 688 46 Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark
689 Batty. Modular relaxed dependencies in weak memory concurrency. In Peter Müller, editor,
690 *Programming Languages and Systems*, pages 599–625, Cham, 2020. Springer International
691 Publishing.
- 692 47 Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In Bart Jacobs,
693 Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Conference on the*
694 *Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA,*
695 *June 12-15, 2014*, volume 308 of *Electronic Notes in Theoretical Computer Science*,
696 pages 273–288. Elsevier, 2014. URL: <https://doi.org/10.1016/j.entcs.2014.10.015>,
697 doi:10.1016/J.ENTCS.2014.10.015.
- 698 48 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming
699 languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL), jan
700 2019. doi:10.1145/3290382.
- 701 49 Damien Pous. Coinduction All the Way Up. In *Thirty-First Annual ACM/IEEE Symposium*
702 *on Logic in Computer Science (LICS 2016)*, New York, United States, July 2016. ACM. URL:
703 <https://hal.archives-ouvertes.fr/hal-01259622>, doi:10.1145/2933575.2934564.
- 704 50 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell.
705 Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8.
706 *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi:10.1145/3158107.
- 707 51 Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur.
708 Promising-arm/risc-v: a simpler and faster operational concurrency model. In *Proceedings of*
709 *the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*,
710 PLDI 2019, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.
711 doi:10.1145/3314221.3314624.
- 712 52 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. Qed at
713 large: A survey of engineering of formally verified software. *Foundations and Trends® in*
714 *Programming Languages*, 5(2-3):102–281, 2019.
- 715 53 Grigore Roşu and Traian Florin Şerbănuță. An overview of the k semantic framework.
716 *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. Membrane

- 717 computing and programming. URL: <http://www.sciencedirect.com/science/article/pii/S1567832610000160>, doi:10.1016/j.jlap.2010.03.012.
- 718
- 719 **54** Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press,
720 USA, 2nd edition, 2012.
- 721 **55** Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics
722 for noninterference with interaction trees. In Karim Ali and Guido Salvaneschi, editors, *37th*
723 *European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023,*
724 *Seattle, Washington, United States*, volume 263 of *LIPICs*, pages 29:1–29:29. Schloss Dagstuhl -
725 Leibniz-Zentrum für Informatik, 2023. URL: <https://doi.org/10.4230/LIPICs.ECOOP.2023.29>,
726 doi:10.4230/LIPICs.ECOOP.2023.29.
- 727 **56** Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer.
728 Conditional contextual refinement. *Proc. ACM Program. Lang.*, 7(POPL):1121–1151, 2023.
729 doi:10.1145/3571232.
- 730 **57** Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter
731 Sewell. CompCertso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3),
732 jun 2013. doi:10.1145/2487241.2487248.
- 733 **58** Li yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce,
734 and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq.
735 *Proc. ACM Program. Lang.*, 4(POPL, Article 51), 2020.
- 736 **59** Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic
737 interpreters. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi:10.1145/3547630.
- 738 **60** Yannick Zakowski, Calvin Beck, Irene Yoon, Ilija Zaichuk, Vadim Zaliva, and Steve Zdancewic.
739 Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program.*
740 *Lang.*, 5(ICFP), aug 2021. doi:10.1145/3473572.
- 741 **61** Junpeng Zha, Hongjin Liang, and Xinyu Feng. Verifying optimizations of concurrent programs
742 in the promising semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference*
743 *on Programming Language Design and Implementation, PLDI 2022*, pages 903–917, New York,
744 NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523734.
- 745 **62** Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer,
746 William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server
747 with interaction trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International*
748 *Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy*
749 *(Virtual Conference)*, volume 193 of *LIPICs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-
750 Zentrum für Informatik, 2021. URL: <https://doi.org/10.4230/LIPICs.ITP.2021.32>, doi:
751 10.4230/LIPICs.ITP.2021.32.
- 752 **63** Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. Fully composable
753 and adequate verified compilation with direct refinements between open modules. *Proc. ACM*
754 *Program. Lang.*, 8(POPL):2160–2190, 2024. doi:10.1145/3632914.
- 755 **64** Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing
756 the LLVM Intermediate Representation for Verified Program Transformations. In *Proc.*
757 *of the ACM Symposium on Principles of Programming Languages (POPL)*, 2012. doi:
758 10.1145/2103621.2103709.
- 759 **65** Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal
760 verification of SSA-based optimizations for LLVM. In *Proc. 2013 ACM SIGPLAN Conference*
761 *on Programming Languages Design and Implementation (PLDI)*, 2013. doi:10.1145/2499370.
762 2462164.

763 **A** Implementation of thread creation

764 As an intermediate representation, LLVM IR does not specify how threads can be created,
 765 this is left to higher-level programming languages and APIs. By itself, our `spawn` syntax is
 766 too low-level to be practical, in particular it does not enforce a consistent view on memory
 767 between the caller thread and the freshly-created one. However, it is parameterized by thread
 768 initialization and cleanup functions whose code is respectively prepended and appended to
 769 the code of the actual task to run. We can use these functionalities to implement more
 770 realistic thread handling semantics⁷. We base the semantics of thread creation and joining
 771 on `thrd_create` and `thrd_join` from the C11 standard library [20]. It is similar to POSIX
 772 `pthread_create` and `pthread_join`, but the interactions between these functions and the
 773 memory model are more clearly specified in the C standard than in the POSIX one.

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

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

⁷ 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.