Structural temporal logic for mechanized program verification

ELEFTHERIOS IOANNIDIS, University of Pennsylvania, United States
YANNICK ZAKOWSKI, ENS Lyon, Inria, France
STEVE ZDANCEWIC, University of Pennsylvania, United States
SEBASTIAN ANGEL, University of Pennsylvania, United States

Mechanized verification of liveness properties for infinite programs with effects and nondeterminism is challenging. Existing temporal reasoning frameworks operate at the level of models such as traces and automata. Reasoning happens at a very low-level, requiring complex nested (co-)inductive proof techniques and familiarity with proof assistant mechanics (e.g., the guardedness checker). Further, reasoning at the level of models instead of program constructs creates a verification gap that loses the benefits of modularity and composition enjoyed by structural program logics such as Hoare Logic. To address this verification gap, and the lack of compositional proof techniques for temporal specifications, we propose `Ticl`, a new structural temporal logic. Using `Ticl`, we encode complex (co-)inductive proof techniques as structural lemmas and focus our reasoning on variants and invariants. We show that it is possible to perform compositional proofs of general temporal properties in a proof assistant, while working at a high level of abstraction. We demonstrate the benefits of `Ticl` by giving mechanized proofs of safety and liveness properties for programs with scheduling, concurrent shared memory, and distributed consensus, demonstrating a low proof-to-code ratio.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program specifications**.

Additional Key Words and Phrases: Formal Verification, Semantics, Temporal Logic, Program Verification, Proof Assistant,Systems Verification

1 Introduction

Mechanized program verification can guarantee that executable code satisfies formal specifications categorized as either liveness or safety properties. Liveness properties ("a good thing happens") include *termination* and *fairness*, as well as *always-eventually* properties. Liveness properties appear in web servers ("the server *always-eventually* replies to requests"), operating systems ("the memory allocator will *eventually* return a memory page", "the scheduler is *fair*") and distributed protocols ("a consensus is *eventually* reached"). Despite their prevalence in computer systems, liveness properties have been understudied compared to safety properties ("a bad thing never happens"), for which numerous general reasoning frameworks and verifications techniques exist [1, 4, 24, 32, 42, 47].

Arguably, the widespread success of mechanized safety verification has been due to the development of program logics that are *compositional* and reason directly over the *structure* of programs. An example is Hoare logic, with its basic construct, the Hoare triple $\{P\}\ c\ \{Q\}$, which specifies that if the precondition P holds before executing the command c, and c terminates, then the postcondition Q will hold afterward. Hoare logic allows one to perform local reasoning by breaking down complex programs into small components, and to verify individual parts without needing to understand the whole. Then, using the *sequence rule*, one can combine triples $\{P\}\ c_1\ \{Q\}$ and $\{Q\}\ c_2\ \{R\}$ to get $\{P\}\ c_1; c_2\ \{R\}$, building bigger proofs from smaller subproofs. Hoare rules are structural: they allow reasoning over standard program constructs like assignment ($x \leftarrow a$), conditionals (**if** (c) **then** a **else** b), and loops (**while** $(c)\ \{b\}$), hiding their semantic interpretations.

Authors' Contact Information: Eleftherios Ioannidis, University of Pennsylvania, Philadelphia, PA, United States, elefthei@cis.upenn.edu; Yannick Zakowski, ENS Lyon, Inria, Lyon, France, yannick.zakowski@inria.fr; Steve Zdancewic, University of Pennsylvania, Philadelphia, PA, United States, stevez@cis.upenn.edu; Sebastian Angel, University of Pennsylvania, Philadelphia, PA, United States, sebastian.angel@cis.upenn.edu.

Unfortunately, this picture could not be more different when it comes to proving liveness properties. While there are very powerful logics for reasoning about general concepts of progress and time, namely *temporal logics* [3, 7, 19, 26, 27, 36], these tend to focus on *semantic models* of program execution. In other words, instead of writing proofs about standard program constructs as shown above, one first models programs as automata or infinite traces and then reasons about these models instead [3, 15, 16, 19, 22, 36, 43]. Mechanized reasoning in these semantic models is arduous, requiring nested induction and coinduction techniques (Section 2) and deep understanding of complex mathematical concepts like the Knaster-Tarski lemma (Appendix A), and the proof assistant's mechanics (e.g., the guardedness checker). Additionally, semantic proofs of liveness do not compose with respect to the *sequence* and *iteration* operators, causing proof scalability issues for large programs.

Contributions: We introduce *Temporal Interaction and Choice Logic* (`ticl`), a novel program logic inspired by *Computation Tree Logic* (CTL) [19] that is designed for modular, mechanized verification of liveness and safety properties. `Ticl` extends CTL with program postconditions, similar to those in Hoare logic. Using `ticl` one can write and prove temporal specifications (e.g., *always, eventually, and always-eventually*) at a high-level of abstraction. `Ticl` proofs *compose* with the *sequence* and *iteration* operators, addressing the long-standing challenge of compositional verification of liveness properties. `Ticl` has three goals:

(1) Combine temporal specifications over finite and infinite traces in one proof system. This part is crucial for supporting composition, as `ticl` needs ways to express postconditions that apply to the return values of terminating programs while also being able to handle programs that run forever.
(2) Close the verification gap between executable programs and the formal models used in temporal logics (e.g., traces and transition systems). `Ticl` achieves this using a new mathematical model of computation that we call `ICTrees`. As a part of the Interaction Trees family [10, 50], `ICTrees` encode programs in different programming languages [11, 29, 30, 39, 40, 49, 54] with support for program extraction and formal transition system semantics.
(3) Develop a library of 50 high-level structural lemmas that proof engineers can readily apply to programs in order to prove liveness properties. These lemmas internalize more than 20,000 lines of complex (co-)inductive proofs, hiding that complexity from the user. To use these lemmas, proof engineers must define their programs using `ICTrees` and then write temporal specifications as `ticl` formulas (Section 6). These specifications are then proved structurally, in a manner similar to Hoare Logic proofs, and without the usual (co-)induction bureaucracy.

We demonstrate that `ticl` is sufficiently expressive to prove meaningful safety and liveness specifications with a small proof-to-program ratio. We use examples spanning sequential, concurrent, and distributed programming: imperative programs with heaps, a round-robin scheduler, concurrent programs with shared memory, and a simple distributed consensus protocol. Our development is formalized in the Rocq proof assistant [46] (formerly known as "Coq"), relying solely on the *uniqueness of identity proofs* axiom (UIP or `eq_rect_eq`). `Ticl` is released under an open-source license[1].

Related Work: Beyond LTL and CTL (Section 8 offers a deeper comparison), step-indexed logical relation frameworks like Iris [4, 24] can prove safety but not liveness properties. More recently, transfinite extensions to step-indexing [44] made it possible to prove *always* properties but not *always-eventually* properties. Certain liveness properties have been studied in a *syntactic* setting [17, 29, 31] but these are limited in expressivity and do not provide a general framework

[1] https://github.com/vellvm/ticl

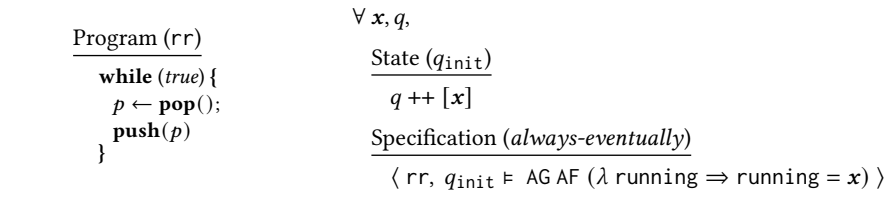

Fig. 1. Example of a round-robin scheduler program (rr). The initial state of the program ($q_{\texttt{init}}$) consists of a designated thread x appended to a list of other threads q. The liveness specification, given in CTL [19], asserts that for all possible designated threads (x) and for all initial lists of other threads (q), thread x will always (AG)-eventually (AF) be popped from the queue (i.e., running = x).

for arbitrary temporal specifications. For example, Fair Operational Semantics [29] are limited to binary *always-eventually* properties, specifically *good* vs. *bad* events, and do not generalize to arbitrary liveness properties. Many deductive verification frameworks for temporal properties, for example Cyclist [45], CoqTLA [9] and the Maude language [33] operate on the semantic level of models, not on the syntactic level of code, missing the advantages of structural program logics.

Limitations: ticl has extensive support for backwards reasoning (systematically weakening a goal specification into smaller subgoals and proving them), less support is included at this point for forward reasoning (strengthening and combining known hypotheses to create new hypotheses). Some support for forward reasoning is offered through custom tactics and inversion lemmas we developed. Still, as we report in the feature table of Figure 14, proving forward reasoning principles for some of ticl's constructions remains open question, which we leave for future work. Ticl also inherits the same limitations of completeness (with respect to specifying liveness properties) found in prior variants of temporal logics [2, 48]. We discuss this in more detail in Section 8.4.

2 Why are liveness properties so challenging to prove?

We illustrate the challenges of formally proving a basic liveness property for a small program. Consider an operating system that maintains a queue of threads with some tasks, and a round-robin scheduler that processes each thread one after the other. The program rr in Figure 1 implements round-robin scheduling—a simple infinite loop removes a thread from the head of the queue and re-inserts it at the end. Our goal is to prove that a thread $\boldsymbol{x}$ will *always-eventually* be scheduled (AG AF using CTL notation [19]).

One approach to mathematical reasoning about infinite programs is to represent them as coinductive trees of events. The infinite loop in Figure 1 unfolds to a coinductive stream of alternating [pop, push, pop . . .] events. Popping removes an element from the *head* of the queue, while pushing appends an element. Applying an infinite stream of alternating pop and push events to the initial queue state, $q_{\texttt{init}}$, results in a coinductive tree of queue states, as shown in Figure 2. Each infinite trace depends on the length of q.

The goal property ("always-eventually x is running") is a nested inductive and coinductive predicate over the coinductive tree in Figure 2. Proving this property requires nested induction on the length of q, and coinduction on each trace. The proof is hard—working directly with trees of traces and low-level induction/coinduction tactics is neither *modular* nor *structural*. The trivial-looking example of Figue 1 requires a non-trivial amount of infrastructure to prove, most of which is not reusable for other programs and specifications. With ticl, proving the example from Figure 1 is reduced to a simple application of the *invariance* rule for while loops (for a preview, see Figure 24).

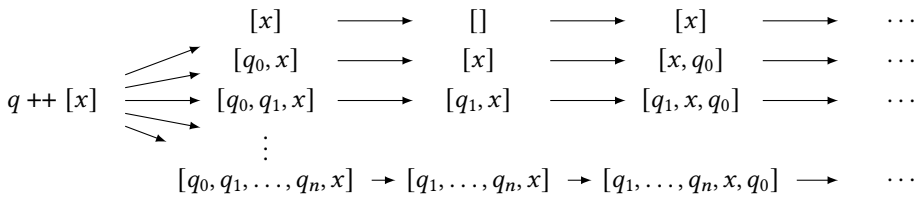

Fig. 2. Coinductive traces for the `rr` program (Figure 1) and all possible initial states q ++ $[x]$.

3 Computational model: `ICTrees`

In this section we introduce a model of computation that we call *Interaction and Choice Trees* (`ICTrees`). This denotational model, inspired by prior works [10, 50], is expressive enough to formalize programming languages with nondeterminism, nontermination, and interaction (we discuss how it relates to prior works in detail in Section 8.1). Further, `ICTrees` have standard combinators like sequencing (bind), iteration and choice and support semantic interpretation [50]. In Sections 6,7 we define the semantics of several programming languages using `ICTrees`, and write example programs in those languages.

3.1 The `ictree` computational model

The `ICTree` coinductive datastructure represents (in-)finite, nondeterministic, and effectful programs. `ICTrees` are defined in Figure 3 using four kinds of nodes: visible event nodes (`Vis`), silent nodes (`Tau`), nondeterministic choice nodes (`Br`), and nodes returning a value (`Ret`).

`Vis` nodes store events ($E \in Type \to Type$) representing interactions of the program with the environment. An event ($e \in E\ X$) is an action, expecting an environment response (X). For example, `Pop` events emitted by the **pop**() command in Figure 1 have type $E_Q\ \mathbb{N}$ (Figure 22) and expect a natural number response ($\mathbb{N}$) representing a thread ID. `Vis` nodes have a child node for every response—in the case of `Pop` there is one child for every possible natural number (Figure 4).

`Br` nodes represent finitary, nondeterministic choice. For example, the program that flips a coin that can be either heads or tails (`flip` $\coloneqq H \oplus T$) denotes to the binary choice (`Br` 2) node shown in Figure 4 (left). These nodes count as a "step" of computation that nondeterministically chooses among n possible continuations (see transition relation in Section 4.1). They are inspired by the equivalent notion of so-called *stepping* branching nodes (`BrS`) used by Choice Trees [10].

`Ret` nodes capture the return value of a terminating computation (`Ret` nodes have no children). `Tau` nodes are *silent* steps, representing internal computation. Their name originates from τ transitions in CCS [35] which indicate an internal action that is separate from process communication. `Tau` nodes are necessary to model programs with control flow that might not terminate (e.g., program `rr` in Figure 1). In the next section we introduce the `ICTrees` transition relation, which transitively closes over `Tau` nodes, making them unobservable to `ticl` formulas (Figure 8). The idea is, we only care about program specifications with respect to their observable behavior. Hiding `Tau` nodes in the transition relation and equational theory—as we will see next—makes the internal behavior of programs (e.g., unobservable control flow) irrelevant. The *stuck* ($\emptyset$) `ICTree` represents the deadlocked program that cannot make progress, and is defined as an infinite chain of `Tau` nodes.

`ICTrees` are *monads*, meaning the monadic composition operations bind ($\gg\!=$) and return (`Ret`) are defined (Figure 3) and satisfy the monad laws (Figure 5). Such sequential composition of both finite and infinite programs is a key property of `ICTrees` that makes them suitable models for `Ticl`, a logic supporting both finite and infinite specifications. Nondeterministic binary choice ($\oplus$)

$$\texttt{ictree} \in (\texttt{Type} \to \texttt{Type}) \to \texttt{Type} \to \texttt{Type}$$

$$\begin{aligned}\texttt{ictree}_{E,X} \overset{\text{coind}}{=}\ & \mid \texttt{Ret}\ (x \in X) && \mid \texttt{Vis}\ (X \in \texttt{Type})\ (e \in E\,X)\ (k \in X \to \texttt{ictree}_{E,X})\\ & \mid \texttt{Tau}\ (t \in \texttt{ictree}_{E,X}) && \mid \texttt{Br}\ (n \in \mathbb{N})\ (k \in \texttt{fin}\ n \to \texttt{ictree}_{E,X})\end{aligned}$$

$$\emptyset \in \texttt{ictree}_{E,X} = \texttt{Tau}\ \emptyset$$

$$\gg\!= \in \texttt{ictree}_{E,X} \to (X \to \texttt{ictree}_{E,Y}) \to \texttt{ictree}_{E,Y}$$

$$(\texttt{Ret}\ x) \gg\!= f = f\ x, \qquad (\texttt{Vis}\ X\ e\ k) \gg\!= f \overset{\text{coind}}{=} \texttt{Vis}\ X\ e\ (\lambda\ (x \in X) \Rightarrow (k\ x) \gg\!= f)$$

$$(\texttt{Tau}\ t) \gg\!= f \overset{\text{coind}}{=} \texttt{Tau}\ (t \gg\!= f), \qquad (\texttt{Br}\ n\ k) \gg\!= f \overset{\text{coind}}{=} \texttt{Br}\ n\ (\lambda\ (i \in \texttt{fin}\ n) \Rightarrow (k\ i) \gg\!= f)$$

$$(x \leftarrow t\ ;;\ k\ x) \in \texttt{ictree}_{E,Y} = (t \in \texttt{ictree}_{E,X}) \gg\!= \lambda\ (x \in X) \Rightarrow (k \in X \to \texttt{ictree}_{E,Y})\ x$$

$$\texttt{iter} \in (I \to \texttt{ictree}_{E,I+R}) \to I \to \texttt{ictree}_{E,R}$$

$$\texttt{iter step i} \overset{\text{coind}}{=} (\texttt{step i}) \gg\!= \lambda\ (lr \in I + R) \Rightarrow \begin{cases}\texttt{Tau}\ (\texttt{iter step i}'), & lr = \texttt{inl}\ i'\\ \texttt{Ret}\ (r), & lr = \texttt{inr}\ r\end{cases}$$

$$\texttt{trigger}\ (e \in E\,X) \in \texttt{ictree}_{E,X} = \texttt{Vis}\ X\ e\ (\lambda\ (x \in X) \Rightarrow \texttt{Ret}\ x)$$

$$\texttt{branch}\ (n \in \mathbb{N}) \in \texttt{ictree}_{E,\texttt{fin}\ n} = \texttt{Br}\ n\ (\lambda\ (i \in \texttt{fin}\ n) \Rightarrow \texttt{Ret}\ i)$$

$$\oplus \in \texttt{ictree}_{E,X} \to \texttt{ictree}_{E,X} \to \texttt{ictree}_{E,X}$$

$$\texttt{l} \oplus \texttt{r} = \texttt{Br}\ \_\left(\lambda\ (i \in \texttt{fin}\ 2) \Rightarrow \begin{cases}\texttt{l}, & i = F_1\\ \texttt{r}, & i = FS\ F_1\end{cases}\right)$$

Fig. 3. Definition of the ICTree datastructure and core ICTree combinators.

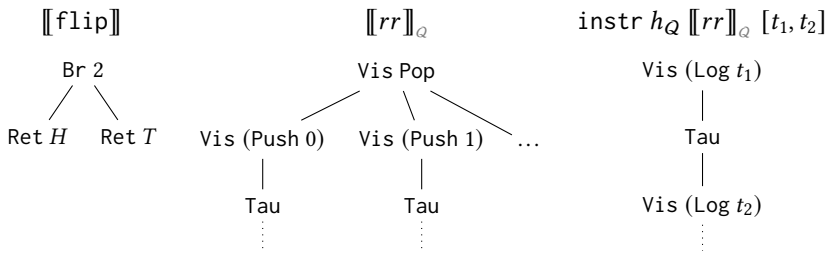

Fig. 4. Example ICTrees denoting programs flip, rr, and the instrumentation of rr with two threads.

is defined by matching on the nondeterminstic result of Br 2. Operations trigger and branch are wrappers around visible event nodes (Vis) and n-ary nondeterministic choice (Br).

Looping programs—both finite and infinite—can be constructed through the iter combinator in Figure 3, which accepts a *stepping function* ($k \in I \to \texttt{ictree}_{E,I+R}$) and seed ($i \in I$). If the stepping function returns a value of type I (iterator), the loop continues with a new iterator. Otherwise, it terminates with a return value of type R (result). Note that iter emits a Tau node every loop repetition, indicating the internal control flow. For example, in Figure 4 (center) we show how program rr (encoded in ICTrees from the code in Figure 1) infinitely repeats after the Tau node.

The addition of Tau nodes makes programs with the same observable behaviors appear different syntactically (e.g., a loop and its unfolding have the same behavior but are syntactically different).

$$\frac{}{t \sim t}\ \textsc{SbRefl} \qquad \frac{t \sim u}{u \sim t}\ \textsc{SbSym} \qquad \frac{t \sim u \qquad u \sim v}{t \sim v}\ \textsc{SbTrans} \qquad \frac{}{\mathtt{Tau}\ t \sim t}\ \textsc{SbTau}$$

$$\frac{t \sim u \qquad (\forall x,\ g\ x \sim k\ x)}{t \gg\!= g \sim u \gg\!= k}\ \textsc{SbBind} \qquad \frac{}{\mathtt{Ret}\ v \gg\!= k\ \sim\ k\ v}\ \textsc{SbBindL} \qquad \frac{}{x \leftarrow t;; \mathtt{Ret}\ x\ \sim\ t}\ \textsc{SbBindR}$$

$$\frac{}{(t \gg\!= k) \gg\!= l\ \sim\ t \gg\!= (\lambda x \Rightarrow k\ x \gg\!= l)}\ \textsc{SbBindAssoc} \qquad \frac{x = y}{\mathtt{Ret}\ x \sim \mathtt{Ret}\ y}\ \textsc{SbRet}$$

$$\frac{\forall x,\ h\ x \sim k\ x}{\mathtt{Vis}\ e\ h \sim \mathtt{Vis}\ e\ k}\ \textsc{SbVis} \qquad \frac{\forall x,\ h\ x \sim k\ x}{\mathtt{Br}\ n\ h \sim \mathtt{Br}\ n\ k}\ \textsc{SbBr}$$

Fig. 5. Equational theory for ICTree with respect to *up-to-tau* equivalence relation (∼). A double inference line indicates a coinductive rule.

$$\mathtt{L}_W \in \mathtt{Type} \to \mathtt{Type} \ :=\ \mid \mathtt{Log}\ (w \in W) \in \mathtt{L}_W\ \mathtt{unit}$$
$$\mathbf{log}\ (w \in W) \in \mathtt{ictree}_{\mathtt{L}_W,\,\mathtt{unit}} \ :=\ \mathtt{trigger}\ \mathtt{Log}\ w$$
$$\mathbf{InstrM}_{S,W} \in \mathtt{Type} \to \mathtt{Type} \ :=\ \mathtt{stateT}\ \mathtt{S}\ \mathtt{ictree}_{\mathtt{L}_W}$$

$$\mathbf{instr} \in (E \rightsquigarrow \mathtt{InstrM}_{S,W}) \to \mathtt{ictree}_E \rightsquigarrow \mathtt{InstrM}_{S,W}$$
$$\mathtt{instr}\ h\ (\mathtt{Ret}\ x)\ s\ =\ \mathtt{Ret}\ (x, s),$$
$$\mathtt{instr}\ h\ (\mathtt{Tau}\ t)\ s\ \overset{\text{coind}}{=}\ \mathtt{Tau}\ (\mathtt{instr}\ h\ t\ s)$$
$$\mathtt{instr}\ h\ (\mathtt{Vis}\ X\ e\ k)\ s\ \overset{\text{coind}}{=}\ (h\ e\ s)\ \gg\!=\ (\lambda\ `(x \in X, s' \in S) \Rightarrow \mathtt{Tau}\ (\mathtt{instr}\ h\ (k\ x)\ s'))$$
$$\mathtt{instr}\ h\ (\mathtt{Br}\ n\ k)\ s\ \overset{\text{coind}}{=}\ \mathtt{Br}\ n\ (\lambda\ (i \in \mathtt{fin}\ n) \Rightarrow \mathtt{instr}\ h\ (k\ i)\ s))$$

Fig. 6. Instrumentation of an $\mathtt{ictree}_E$ produces the monad $\mathtt{InstrM}_{S,W}$, that "remembers" the temporal order of events E via observations $\mathtt{L}_W$.

This is undesirable because later when we introduce `ticl` formulas we will want to say things like: a program with a finite loop terminates if and only if the equivalent program by unrolling the loop terminates. To make such reasoning possible, we define a notion of ICTree equivalence, called *up-to-tau* equivalence, that ICTrees inherits from prior Interaction Tree works [10, 50]. Up-to-tau equivalence is invariant to (inductively) adding or removing taus on either side; it equates programs with the same observable behavior. For instance, it *does not* equate the stuck $\emptyset$ tree with any terminating tree, such as `Ret` x, but it does equate `Ret x` with `Tau (Ret x)`, since both terminate and return the same `x`. Up-to-tau equivalence (∼) is defined coinductively in Figure 5, along with some useful equations. With this equational theory, programs can be simplified using the monad laws, removing taus, unfolding loops, and more.

3.2 ICTree semantics and instrumentation

Up until this point we have treated ICTrees as abstract syntax. We will now assign to them semantic meaning.

The semantic meaning of events is given by a semantic handler $h : E \rightsquigarrow M$, where M is a monad compatible with the ICTree structure. For example, the "`read` x" and "`write` $x\ v$" events could be interpreted as functions defined to operate in a shared state monad, where the state

type (S) is a map from variables to their values. There, we would have $h_S(\texttt{read}\ x)$ defined by the implementation $\lambda\ (s \in S) \Rightarrow \texttt{Ret}\ (s, s[x])$. That is, the function that takes the current state s and returns it, along with the value of x in that state. The operation $h_S(\texttt{write}\ x\ v)$ would be defined by the function $\lambda\ (s \in S) \Rightarrow \texttt{Ret}\ ((x \hookrightarrow v) \cup s, ())$. That is, the function that takes the current state and returns the modified state along with the unit value (signifying that write does not itself return a value). In general, the monad M of a handler must be compatible with the `iter` and $\oplus$ constructs required by `ICTrees`. In the case above, the types of the `read` and `write` operations are of the form: $S \to \texttt{ictree}_{\texttt{void},\ (S \times X)}$—i.e., they are functions from the starting state to a (potentially diverging) `ictree`. This type is an instance of the *state monad transformer*: $\texttt{stateT S (ictree}_{\texttt{void}})$.

However, interpretation is insufficient for temporal reasoning—in addition to the result of a program, formulas may specify the temporal order of events. This is analoguous to big-step vs. small-step semantics: `ICTree` interpretation ($\texttt{ictree}_E \rightsquigarrow \texttt{stateT S ictree}_{\texttt{void}}$) erases the small-step event information in the tree's `Vis` nodes, returning a big-step reduction in the form of the partial function $\texttt{stateT S ictree}_{\texttt{void}}$. At that point the provenance of events has been erased.

We address the loss of provenance by introducing the notion of *instrumentation*. Intuitively, instrumentation defines which events and their accompanying (ghost) state are considered to be relevant for the `ticl` specifications. Event instrumentation interprets an event ($e \in E\ X$) over a slightly different state monad ($\texttt{stateT S ictree}_{\texttt{L}_W}$) that we call the instrumentation monad ($\texttt{InstrM}_{S,W}$ in Figure 6). Our goal is to interpret events E, leaving behind a trace of *observation* events of type $\texttt{L}_W$. Observation events, or *log events*, are themselves uninterpreted events—the environment response is always of type `unit`. Observations are left behind by event interpretation and signify a "memory" of an environment interaction, recorded as an observation of type W. Log events can be erased without altering the semantics of the program. Values of type W encode auxilary (ghost) state that can be queried by `ticl` formulas. The specification author is free to pick an arbitrary type W to observe and reason about.

For example, program (`rr`) from Figure 1 denotes to the `ICTree` in the middle of Figure 4. The instrumentation handler for queues (h_Q defined later in Figure 22) logs the result of `Pop` events, while it interprets away `Push` events. As a result, instrumenting `rr` results in the infinite trace on the right of Figure 4, showing every value popped from the head of the queue. To make salient the use of instrumentation handlers, we provide several examples in Sections 6.1 and 7.1–7.3.

4 Temporal specifications: `Ticl`

`Ticl` is defined using a ternary entailment relation $\langle\ t,\ w \vDash\ p\ \rangle$, which can be read as "program t satisfies formula p, starting at a world w". Our goal in this section is to define the necessary components of `ticl` entailment; t is an `ICTree` describing (in-)finite, effectful, nondeterministic programs, w captures the initial state of the external *world*, and p is a `ticl` formula.

4.1 Kripke transition relation

Temporal logics are commonly defined over traces or transition systems, stepping from one "world" to another. This will also be true of `ticl`, so we review this concept and describe how transitions apply to `ICTrees`.

A Kripke *world* ($\mathcal{W}_E$), parametrized by an event type E, is a datatype that "remembers" the status of the program, for instance, a past observation and/or its return value. A `Pure` world indicates that no event has been observed yet. A world "`Obs` $e\ v$" remembers the last observed event ($e \in E\ X$) and the response it obtained from interacting with its environment ($v \in X$). A world "`Val` x" captures the return value (x) of a pure program that has terminated. A world

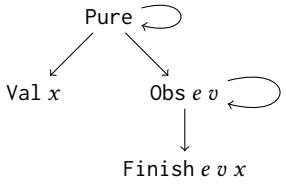

Fig. 7. Transitions between Kripke worlds for `ICTrees`.

$$\mathcal{W}_E \in \texttt{Type} = \texttt{Pure} \mid \texttt{Obs}\ e\ v \mid \texttt{Val}\ x \mid \texttt{Finish}\ e\ v\ x$$

$$\texttt{not\_done Pure} \qquad \texttt{not\_done (Obs}\ e\ v\texttt{)} \qquad \frac{(P_X \in X \to \mathbb{P}) \qquad P_X\ x}{\texttt{done\_with}\ P_X\ \texttt{(Val}\ x\texttt{)}}$$

$$\frac{(P_X \in E\ Y \to Y \to X \to \mathbb{P}) \qquad P x\ e\ v\ x}{\texttt{done\_with}\ P_X\ \texttt{(Finish}\ e\ v\ x\texttt{)}} \qquad \frac{[t,\ w] \mapsto [t',\ w']}{[\texttt{Tau}\ t,\ w] \mapsto [t',\ w']} \qquad \frac{\texttt{not\_done}\ w \qquad 0 \le i < n}{[\texttt{Br}\ n\ k,\ w] \mapsto [k\ i,\ w]}$$

$$\frac{\texttt{not\_done}\ w}{[\texttt{Vis}\ e\ k,\ w] \mapsto [k\ v,\ \texttt{Obs}\ e\ v]} \qquad [\texttt{Ret}\ x,\ \texttt{Pure}] \mapsto [\emptyset,\ \texttt{Val}\ x] \qquad [\texttt{Ret}\ x,\ \texttt{Obs}\ e\ v] \mapsto [\emptyset,\ \texttt{Finish}\ e\ v\ x]$$

Fig. 8. Kripke transition relation for ICTrees ($\mapsto$) and world predicates (not_done, done_with). P_X is a type-theoretic ticl postcondtion (described in Section 4.3).

"Finish $e\ v\ x$" captures the return value (x) of an effectful program that terminated, and the last event ($e \in E\ X$) and response ($v \in X$). Worlds ($\mathcal{W}_E$) are divided into done worlds (Val and Finish) and not_done worlds (Pure and Obs), indicating whether a program terminated or is still running. The predicate done_with (Figure 8) enforces a postcondition (P_X) on a done world and will be used to define ticl postconditions in Section 4.3.

Figure 8 defines the Kripke transition relation for ICTrees ($[t,\ w] \mapsto [t',\ w']$). This is an irreflexive binary relation over pairs of $\texttt{ictree}_E$ and worlds ($\mathcal{W}_E$), inductively defined over Tau nodes. Transitions only make sense in not_done worlds since done worlds represent programs that have already terminated. Within not_done worlds, an ICTree can either transition from a Pure world onto another Pure world, it can observe an event and its result with "Obs $e\ v$", or it can terminate with "Val x". An ICTree program can transition from "Obs $e\ v$" onto another "Obs $e'\ v'$" or it can terminate with "Finish $e\ v\ x$". We summarize these transitions in Figure 7.

There are two goals informing our definition of the ICTree transition relation in Figure 8: (1) respecting the up-to-tau equivalence (Section 3.1); and (2) respecting the monad composition laws (Figure 5). The first goal is achieved by defining the transition relation inductively over Tau nodes. The intuition is that ticl observes the external behavior of programs, and internal steps should not change the outcome of a ticl specification. Consequently, up-to-tau equivalent programs should satisfy the same ticl formulas. This is a restatement of the well-known result that strong bisimulation preserves CTL properties [18], which we are able to formally prove in Section 4.5. The second goal—respecting the laws of monadic composition—is achieved through the lemmas in Figure 9. These lemmas describe how the composition of two programs t and k ($x \leftarrow t;;\ k\ x$) transition. Either t transitions to t' and $x \leftarrow t;;\ k\ x$ transitions to $x \leftarrow t';;\ k\ x$, or t terminates with return value x (and possibly an observation e) and then the continuation ($k\ x$) transitions (t'). These lemmas allow us to break proofs of bind transitions into smaller subproofs by case matching on t, a technique we use thoroughly in our development.

4.2 Syntax of ticl

A crucial question that ticl must answer is how to handle both infinite and terminating program specifications. Temporal logics like LTL and CTL assume infinite traces, whereas finite LTL assumes finite traces [14, 20, 36]. Our goal is a specification language that works for both finite and infinite

$$\frac{[t,\ w] \mapsto [t',\ w'] \qquad \texttt{not\_done}\ w'}{[x \leftarrow t;;\ k\ x,\ w] \mapsto [x \leftarrow t';;\ k\ x,\ w']} \qquad \frac{[t,\ w] \mapsto [\emptyset,\ \texttt{Val}\ x] \qquad [k\ x,\ w] \mapsto [t',\ w']}{[x \leftarrow t;;\ k\ x,\ w] \mapsto [t',\ w']}$$

$$\frac{[t,\ w] \mapsto [\emptyset,\ \texttt{Finish}\ e\ v\ x] \qquad [k\ x,\ w] \mapsto [t',\ w']}{[x \leftarrow t;;\ k\ x,\ w] \mapsto [t',\ w']}$$

Fig. 9. Lemmas connecting Kripke transitions ($\mapsto$) to `ICTree` composition.

programs *compositionally*. For instance, an "always" proof should be broken up into a finite "until" prefix and an infinite "always" suffix.

The `ticl` syntax (Figure 10) is inspired by CTL [19] using the same path-quantified temporal operators, with some notable differences. There are two syntactic categories in `ticl`: *prefix* formulas (φ) that represents predicates on the prefix of a tree (or on infinite trees), and *suffix* (ψ_X) formulas that represent postconditions on terminating trees. Suffix formulas (ψ_X) reference prefix formulas (φ) on the left-hand side argument of their binary temporal operators (AN, AU, EN, EU). This is reasonable, as the formula on the left must be satisfied before the one on the right. Due to their appearance on the left side of temporal operators we also refer to prefix formulas as *left* (L) formulas and to suffix formulas as *right* (R). We assign meaning to formulas with the two ternary entailment relations $\vDash_L$ and $\vDash_L$ in Definition 1, and overload the notation $\vDash_{LR}$ to indicate we refer to both $\vDash_L$ and $\vDash_R$.

The dual syntax is novel compared to LTL, CTL and TLA [53]. To motivate the dual syntax, consider the alternative—what if we only chose one syntactic class of `ticl` formulas, either suffix (ψ_X) formulas or prefix formulas (φ). If we only have suffix formulas then every program (including infinite programs) must have a postcondition. The only reasonable postcondition for an infinite program is $\bot$, a choice made by partial correctness program logics prohibiting sound proofs of liveness. If we only have prefix formulas we lose program postconditions, and by extension, sequential proof composition. At the risk of jumping ahead, $\langle\ t,\ w \vDash_R\ \mathsf{AF}\ \mathsf{AX}\ \mathsf{done}\ \mathcal{R}\ \rangle$ means that program t will eventually terminate satisfying postcondition $\mathcal{R}$. As we will see shortly in Section 5, a goal $\langle\ x \leftarrow t;;\ k\ x,\ w \vDash_L\ \mathsf{AF}\ \varphi\ \rangle$ can be broken into two subgoals $\langle\ t,\ w \vDash_R\ \mathsf{AF}\ \mathsf{AX}\ \mathcal{R}\ \rangle$ and $\forall\ x\ w, \mathcal{R}\ x\ w \rightarrow \langle\ k\ x,\ w \vDash_L\ \mathsf{AF}\ \varphi\ \rangle$, where postcondition $\mathcal{R}$ specifies the codomain of t and the domain of its continuation (k). We therefore need formulas recognizing infinite programs (prefixes) and formulas for finite programs with postconditions (suffixes), and a way to compose them.

The syntax and semantics of CTL [20] coincide with prefix formulas. The syntax and semantics of suffix formulas closely resemble those of finite LTL [14]. Moreover, prefix and suffix formulas have different structural lemmas with respect to sequential composition and iteration. For example, if t can run forever, so can $x \leftarrow t;;\ k\ x$, for any k. However, if t terminates with postcondition $\mathcal{R}$ and $x \leftarrow t;;\ k\ x$ runs forever, it must be because the continuation k runs forever starting at $\mathcal{R}$. We revisit this type of lemmas in Section 5.1.

A notational difference of `ticl` with CTL is the *next* operators AN and EN are binary, unlike the AX and EX operators of CTL which are unary. We reclaim their unary versions using syntactic notations (see Figure 10). We elaborate on the comparison of `ticl` with CTL in Section 8.2.

4.3 Semantics of `ticl` formulas

`Ticl` is defined using binary and unary operators. The meaning of `ticl` temporal operators is indicated by their two letters. For the first letter, "A" stands for *all paths* and "E" stands for *exists a path*; these are the same path quantifiers from CTL [19]. For the second letter, "N" stands for "Next". For example, in the binary operator $\varphi\ \mathsf{AN}\ \varphi'$, the formula φ must hold now, and φ' must hold in every possible single step. In contrast, in $\varphi\ \mathsf{EN}\ \varphi'$ ("exists next"), the formula φ must hold

$$
\begin{array}{l}
\varphi, \varphi' ::= \ \mathsf{now}\ (P \in \mathcal{W}_E \to \mathbb{P}) \\
\quad \mid\ \varphi\ \mathsf{AN}\ \varphi' \\
\quad \mid\ \varphi\ \mathsf{EN}\ \varphi' \\
\quad \mid\ \varphi\ \mathsf{AU}\ \varphi' \\
\quad \mid\ \varphi\ \mathsf{EU}\ \varphi' \\
\quad \mid\ \mathsf{AG}\ \varphi \\
\quad \mid\ \mathsf{EG}\ \varphi \\
\quad \mid\ \varphi \wedge \varphi' \\
\quad \mid\ \varphi \vee \varphi'
\end{array}
\qquad
\begin{array}{l}
\psi_X, \psi'_X ::= \\
\quad \mathsf{done}\ (P_X \in X \to \mathcal{W}_E \to \mathbb{P}) \\
\quad \mid\ \varphi\ \mathsf{AN}\ \psi_X \\
\quad \mid\ \varphi\ \mathsf{EN}\ \psi_X \\
\quad \mid\ \varphi\ \mathsf{AU}\ \psi_X \\
\quad \mid\ \varphi\ \mathsf{EU}\ \psi_X \\
\quad \mid\ \psi_X \wedge \psi'_X \\
\quad \mid\ \psi_X \vee \psi'_X
\end{array}
\qquad
\begin{array}{lcl}
\top & = & \mathsf{now}\ (\lambda\ \_.\top) \\
\bot & = & \mathsf{now}\ (\lambda\ \_.\bot) \\
\Tau & = & \mathsf{done}\ (\lambda\ \_\ \_.\top) \\
\Vdash & = & \mathsf{done}\ (\lambda\ \_\ \_.\bot) \\
\mathsf{AX}\ p & = & \top\ \mathsf{AN}\ p \\
\mathsf{EX}\ p & = & \top\ \mathsf{EN}\ p \\
\mathsf{AF}\ p & = & \top\ \mathsf{AU}\ p \\
\mathsf{EF}\ p & = & \top\ \mathsf{EU}\ p
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathsf{pure} & = & \mathsf{now}\ (\lambda\ w.\ w = \mathit{Pure}) & \mathsf{val}\ p & = & \mathsf{done}\ (\lambda\ x\ w.\ w = \mathtt{Val}\ x \wedge p\ x) \\
\mathsf{obs}\ p & = & \mathsf{now}\ (\lambda\ w.\ w = \mathtt{Obs}\ e\ v\ \wedge\ p\ e\ v) & \mathsf{finish}\ p & = & \mathsf{done}\ (\lambda\ x\ w.\ w = \mathtt{Finish}\ e\ v\ x\ \wedge\ p\ x\ e\ v) \\
\mathsf{done}_=\ x\ w & = & \mathsf{done}\ (\lambda\ x'\ w'.\ w = w' \wedge x = x') & & &
\end{array}
$$

Fig. 10. Syntax of `ticl` prefix formulas (φ), suffix formulas (ψ_X), and useful syntactic notations.

$$
\begin{array}{lcl}
\mathsf{can\_step}\ t\ w \in \mathbb{P} & = & \exists\ t', w',\ [t,\ w] \mapsto [t',\ w'] \\
\mathsf{anc}\ P\ Q\ t\ w \in \mathbb{P} & = & P\ t\ w\ \wedge\ \mathsf{can\_step}\ t\ w\ \wedge\ \forall\ t', w',\ [t,\ w] \mapsto [t',\ w'] \to Q\ t'\ w' \\
\mathsf{enc}\ P\ Q\ t\ w \in \mathbb{P} & = & P\ t\ w\ \wedge\ \exists\ t', w',\ [t,\ w] \mapsto [t',\ w']\ \wedge\ Q\ t'\ w' \\
\mathsf{agc}\ P\ Q\ t\ w \in \mathbb{P} & = & \mathsf{gfp}\ (\mathsf{anc}\ P)\ Q\ t\ w \\
\mathsf{egc}\ P\ Q\ t\ w \in \mathbb{P} & = & \mathsf{gfp}\ (\mathsf{enc}\ P)\ Q\ t\ w
\end{array}
$$

$$
\frac{Q\ t\ w}{\mathsf{auc}\ P\ Q\ t\ w}
\qquad
\frac{\mathsf{anc}\ P\ (\mathsf{auc}\ P\ Q\ t\ w)}{\mathsf{auc}\ P\ Q\ t\ w}
\qquad
\frac{Q\ t\ w}{\mathsf{euc}\ P\ Q\ t\ w}
\qquad
\frac{\mathsf{enc}\ P\ (\mathsf{euc}\ P\ Q\ t\ w)}{\mathsf{euc}\ P\ Q\ t\ w}
$$

Fig. 11. *Next* (anc and enc), *globally* (agc and egc) and *until* (auc and euc) higher-order predicates.

now and there exists a single step that satisfies φ'. "U" stands for "Until". For example, in $\varphi\ \mathsf{AU}\ \psi_X$, φ must hold in all paths until eventually ψ_X holds, then φ does not have to hold any longer. The "G" stands for "Globally", as in the formula under this operator must hold *forever*. For example, $\mathsf{EG}\ \varphi$. Using syntactic notations we define "F" as "Finally", a unary version of the inductive "U" that has no left-hand requirement. In CTL "X" stands for "neXt" but in `ticl` "X" is simply a unary version of the binary next "N", with no left-hand formula. Finally, the base formulas now and done are shallow predicates of the metalanguage and apply to the current world ($\mathcal{W}_E$) each time.

Before jumping into the semantics of `ticl` formulas ($\vDash_{LR}$) we must first define the *shallow* predicates of Figure 11. Definitions anc, enc, agc, egc, auc, and euc are *higher-order predicates* in type theory: they take predicates of type $\mathtt{ictree}_{E,\ X} \to \mathcal{W}_E \to \mathbb{P}$ as arguments and transport them under their modal operator to get "future" predicates of the same type. For example, anc means "forall-next", enc means "exists-next", agc means "forall-globally", auc means "forall-until", etc.

The basic predicates are anc and enc, connecting predicates P, Q to the `ICTree` transition relation in Figure 8. All other predicates are (co-)inductively defined in terms of anc and enc in Figure 11. One difference of our definition compared to temporal logics such as CTL is the restriction `can_step` on *forall-next* (anc). Predicate `can_step` asserts the existence of at least one transition and is crucial to prove the soundness of `ticl`. Because the transition relation is not left-total, and allows for stuck states ($\emptyset$), omitting `can_step` allows $\langle\ \emptyset,\ w \vDash_L\ \mathsf{AX}\ \bot\ \rangle$ to be provable in one step—by introducing the hypothesis $[\emptyset,\ w] \mapsto [t',\ w']$ and concluding the proof by contradiction. Predicate `can_step` prohibits vacuously proving statements by asserting that $\emptyset$ can transition.

$$\left.\begin{array}{l} [\![\varphi]\!]_{\mathsf{L}} \in \forall X,\ \mathtt{ictree}_{E,X} \to \mathcal{W}_E \to \mathbb{P}, \quad [\![\psi_{\mathsf{X}}]\!]_{\mathsf{R}} \in \mathtt{ictree}_{E,X} \to \mathcal{W}_E \to \mathbb{P} \end{array}\right\} \text{denotations to shallow predicates}$$

$$\left.\begin{array}{l} [\![\mathsf{now}\ P]\!]_{\mathsf{L}} = \lambda\ \_\ w.\ \mathtt{not\_done}\ w \wedge P\ w, \quad [\![\mathsf{done}\ P_{\mathsf{X}}]\!]_{\mathsf{R}} = \lambda\ \_\ w.\ \mathtt{done\_with}\ P x\ w \end{array}\right\} \text{Base case predicates}$$

$$\left.\begin{array}{l} [\![\varphi\ \mathsf{AN}\ \varphi']\!]_{\mathsf{L}} = \mathsf{anc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\varphi']\!]_{\mathsf{L}}, \quad [\![\varphi\ \mathsf{EN}\ \varphi']\!]_{\mathsf{L}} = \mathsf{enc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\varphi']\!]_{\mathsf{L}} \\ [\![\varphi\ \mathsf{AN}\ \psi_{\mathsf{X}}]\!]_{\mathsf{R}} = \mathsf{anc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\psi_{\mathsf{X}}]\!]_{\mathsf{R}}, \quad [\![\varphi\ \mathsf{EN}\ \psi_{\mathsf{X}}]\!]_{\mathsf{R}} = \mathsf{enc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\psi_{\mathsf{X}}]\!]_{\mathsf{R}} \end{array}\right\} \textit{Next} \text{ operators}$$

$$\left.\begin{array}{l} [\![\varphi\ \mathsf{AU}\ \varphi']\!]_{\mathsf{L}} = \mathsf{auc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\varphi']\!]_{\mathsf{L}}, \quad [\![\varphi\ \mathsf{EU}\ \varphi']\!]_{\mathsf{L}} = \mathsf{euc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\varphi']\!]_{\mathsf{L}} \\ [\![\varphi\ \mathsf{AU}\ \psi_{\mathsf{X}}]\!]_{\mathsf{R}} = \mathsf{auc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\psi_{\mathsf{X}}]\!]_{\mathsf{R}}, \quad [\![\varphi\ \mathsf{EU}\ \psi_{\mathsf{X}}]\!]_{\mathsf{R}} = \mathsf{euc}\ [\![\varphi]\!]_{\mathsf{L}}\ [\![\psi_{\mathsf{X}}]\!]_{\mathsf{R}} \end{array}\right\} \textit{Until} \text{ operators (inductive)}$$

$$\left.\begin{array}{l} [\![\mathsf{AG}\ \varphi]\!]_{\mathsf{L}} = \mathsf{agc}\ [\![\varphi]\!]_{\mathsf{L}}, \qquad [\![\mathsf{EG}\ \varphi]\!]_{\mathsf{L}} = \mathsf{enc}\ [\![\varphi]\!]_{\mathsf{L}} \end{array}\right\} \textit{Globally} \text{ operators (coinductive)}$$

$$\left.\begin{array}{l} [\![p \wedge q]\!]_{\mathsf{LR}} = \lambda\ t\ w.\ [\![p]\!]_{\mathsf{LR}}\ t\ w \wedge [\![q]\!]_{\mathsf{LR}}\ t\ w, \quad [\![p \vee q]\!]_{\mathsf{LR}} = \lambda\ t\ w.\ [\![p]\!]_{\mathsf{LR}}\ t\ w \vee [\![q]\!]_{\mathsf{LR}}\ t\ w \end{array}\right\} \text{Propositional operators}$$

Fig. 12. `Ticl` formula denotations ($[\![\_]\!]_{\mathsf{LR}}$) defined by induction on φ and ψ_{X}.

Path induction for *until* operators (Figure 12) is implemented by the inductive, higher-order predicates auc and euc (Figure 11). There are two cases. The base case asserts Q holds, and the inductive case asserts P holds now, while auc P Q (or euc P Q) holds next. Path coinduction (*always*) is implemented by the agc and egc greatest fixpoints. The `gfp` operator in Figure 11 and the associated machinery for completing coinductive proofs, namely up-to-principles [38], are presented in detail in Appendix A.

The semantics of `ticl` formulas—denoted to type theory—are defined by induction on formulas φ and ψ_{X} in Figure 12. The `ticl` entailment relations then become simple predicate applications to an `ICTree` t and world w as stated in Definition 1.

Definition 1 (Ticl Entailment).

$$\langle\ t,\ w \vDash_L\ \varphi\ \rangle = [\![\varphi]\!]_{\mathsf{L}}\ t\ w, \quad \langle\ t,\ w \vDash_R\ \psi_{\mathsf{X}}\ \rangle = [\![\psi_{\mathsf{X}}]\!]_{\mathsf{R}}\ t\ w$$

4.4 `Ticl` formula equivalence

`Ticl` entailments $\langle\ t,\ w \vDash_{LR}\ p\ \rangle$ are type-theoretic propositions ($\mathbb{P}$) that form a complete lattice $(\mathbb{P}, \to)$. Consequently, denotations of `ticl` formulas ($[\![p]\!]_{\mathsf{LR}} \in \mathtt{ictree}_{E,X} \to \mathcal{W}_E \to \mathbb{P}$) which are type-theory predicates, also form a complete lattice, with respect to the pointwise implications $\Rightarrow_L$ and $\Rightarrow_R$ in Definition 2 (shown below). Taking an implication in both directions introduces an equivalence relation on `ticl` formulas ($\Leftrightarrow_{LR}$). Two `ticl` formulas p, q are (semantically) equivalent ($p \Leftrightarrow_{LR} q$) when for all trees (t) and worlds (w), $\langle\ t,\ w \vDash_{LR}\ p\ \rangle$ if and only if $\langle\ t,\ w \vDash_{LR}\ q\ \rangle$.

Definition 2 (Partial order and equivalence).

$$\varphi \Rightarrow_L \varphi' = \forall\ t, w,\ \langle\ t,\ w \vDash_L\ \varphi\ \rangle \to \langle\ t,\ w \vDash_L\ \varphi'\ \rangle \qquad \varphi \Leftrightarrow_L \varphi' = \varphi \Rightarrow_L \varphi' \text{ and } \varphi' \Rightarrow_L \varphi$$

$$\psi_{\mathsf{X}} \Rightarrow_R \psi'_{\mathsf{X}} = \forall\ t, w,\ \langle\ t,\ w \vDash_R\ \psi_{\mathsf{X}}\ \rangle \to \langle\ t,\ w \vDash_R\ \psi'_{\mathsf{X}}\ \rangle \qquad \psi_{\mathsf{X}} \Leftrightarrow_R \psi'_{\mathsf{X}} = \psi_{\mathsf{X}} \Rightarrow_R \psi'_{\mathsf{X}} \text{ and } \psi'_{\mathsf{X}} \Rightarrow_R \psi_{\mathsf{X}}$$

Now that we have a notion of formula equivalence, building a library of useful (in-)equalities enables fluent proof manipulation. For example, if $p \Leftrightarrow_{LR} q$ and the goal is $\langle\ t,\ w \vDash_{LR}\ p\ \rangle$, we can rewrite it to $\langle\ t,\ w \vDash_{LR}\ q\ \rangle$ instead. Similarly under `ticl` operators: $\langle\ t,\ w \vDash_{LR}\ \mathsf{AX}\ p\ \rangle$ is equivalent to $\langle\ t,\ w \vDash_{LR}\ \mathsf{AX}\ q\ \rangle$, $\langle\ t,\ w \vDash_{LR}\ r \wedge q\ \rangle$ to $\langle\ t,\ w \vDash_{LR}\ r \wedge q\ \rangle$, and so on. Some useful (in-)equalities in the `ticl` library are shown in Figure 13. We elide the boolean algebra laws for space.

4.5 `ICTree` equivalence under `Ticl` entailment

Let us take stock of what we have achieved so far in this section and what our remaining goals are. We defined two ternary entailment relations over `ICTrees` and worlds, $\langle\ t,\ w \vDash_L\ \varphi\ \rangle$ and $\langle\ t,\ w \vDash_R\ \psi_{\mathsf{X}}\ \rangle$, giving meaning to `ticl` formulas. We established a rewriting system over `ticl`

p AN q	$\Rightarrow_{LR}$ p EN q	(AN-weaken)	p AU q	$\Leftrightarrow_{LR}$ $q \vee (p$ AN p AU $q)$	(AU-unfold)
p AU q	$\Rightarrow_{LR}$ p EU q	(AU-weaken)	p EU q	$\Leftrightarrow_{LR}$ $q \vee (p$ EN p EU $q)$	(EU-unfold)
AG φ	$\Rightarrow_{L}$ EG φ	(AG-weaken)	AG φ	$\Leftrightarrow_{L}$ φ AN AG φ	(AG-unfold)
p AN q	$\Rightarrow_{LR}$ p AU q	(AN-until)	EG φ	$\Leftrightarrow_{L}$ φ EN EG φ	(EG-unfold)
p EN q	$\Rightarrow_{LR}$ p EU q	(EN-until)	p AU q	$\Leftrightarrow_{LR}$ p AU p AU q	(AU-idem)
AG φ	$\Rightarrow_{L}$ φ	(AG-M)	p EU q	$\Leftrightarrow_{LR}$ p EU p EU q	(EU-idem)
EG φ	$\Rightarrow_{L}$ φ	(EG-M)	EG EG φ	$\Leftrightarrow_{L}$ EG φ	(EG-idem)
EG $(\varphi \wedge \varphi')$	$\Rightarrow_{L}$ EG $\varphi \wedge EG\varphi'$	(EG-and)	AG AG φ	$\Leftrightarrow_{L}$ AG φ	(AG-idem)
AG $\varphi \vee AG\varphi'$	$\Rightarrow_{L}$ AG $(\varphi \vee \varphi')$	(AG-or)	AG $(\varphi \wedge \varphi')$	$\Leftrightarrow_{L}$ AG $\varphi \wedge AG\varphi$	(AG-and)
EG $\varphi \vee EG\varphi'$	$\Rightarrow_{L}$ EG $(\varphi \vee \varphi')$	(EG-or)			

Fig. 13. Representative ticl formula implications and equivalences.

formulas ($\varphi \Leftrightarrow_L \varphi'$ and $\psi_{\mathsf{X}} \Leftrightarrow_R \psi'_{\mathsf{X}}$) and proved equations useful for simplifying formulas under entailment. We switch our attention back to programs (t) and remind ourselves of up-to-guard equivalence of programs in Section 3.1. To prove a specification $\langle\ t,\ w \vDash_{LR}\ p\ \rangle$ sometimes it is convenient to simplify the formula (p), but sometimes it is convenient to simplify the program t by substituting it with an equivalent program u (where $t \sim u$).

For example, for the monadic bind simplification from Section 3.1 ($x \leftarrow \mathtt{Ret}\ x;; k\ x \sim k\ x$) it seems intuitive that the following goals are also equivalent $\langle\ x \leftarrow \mathtt{Ret}\ x;; k\ x,\ w \vDash_{LR}\ p\ \rangle \Leftrightarrow \langle\ k\ x,\ w \vDash_{LR}\ p\ \rangle$. This intuition is correct and we were able to mechanize the well-known proof that (strong) bisimulation preserves temporal properties [18]. At its core, the proof relies on the following lemma relating Kripke transitions and program equivalence. The transition relation $[t,\ w] \mapsto [t',\ w']$ itself is *not* up-to-tau invariant—it is easy to find a counter-example with a Tau *n*ode in the middle of the tree. However, Lemma ExEquiv (shown below) is a weaker version of up-to-tau invariance for transitions that *is* provable:

Lemma 3 (ExEquiv).
$\forall s,\ t,\ w,\ w',\ (s \sim t) \rightarrow ([s,\ w] \mapsto [s',\ w']) \rightarrow \exists\ t',\ [t,\ w] \mapsto [t',\ w'] \wedge\ (s' \sim t')$

By using lemma ExEquiv and by induction on the structure of ticl formulas, we are able to prove that rewriting with up-to-tau equivalence ($\sim$) under ticl entailment ($\vDash_{L,R}$) is correct, namely:

Theorem 4 (Up-to-tau equivalence preserves ticl formulas).
$\forall t,\ u,\ w,\ p,\ t \sim u \rightarrow \langle\ t,\ w \vDash_{LR}\ p\ \rangle \rightarrow \langle\ u,\ w \vDash_{LR}\ p\ \rangle$

The property of *up-to-tau invariance* enables equational reasoning (Figure 5) in conjunction with structural proof techniques (next Section 5) resulting in a remarkably flexible proof system.

5 Structural lemmas for ICTree

The equational theories of ticl formulas ($\Leftrightarrow_{LR}$) and ICTrees ($\sim$) allow us to simplify a specification such as $\langle\ t,\ w \vDash_{L}\ \mathsf{AG}\ \varphi\ \rangle$, but are insufficient to fully prove it. The inequalities of Figure 13 unfold the "always" operator (AG $\varphi \Leftrightarrow_L\ \varphi$ AN AG φ) but there will always be an AG φ proof obligation left over. In this section we give structural lemmas connecting ICTree composition ($\gg\!=$, $\oplus$) and iteration (iter) to ticl operators (AU, EU, EG and EG) allowing us to fully prove ticl specifications.

The lemmas in this section *internalize* low-level (co-)inductive proofs to simple structural lemmas over ICTrees. As ICTrees form a denotational basis for many programming languages, the lemmas in this section form a logical basis for many temporal logics defined over those languages. In the

	Prefix (φ)						Suffix (ψ_X)			
	AN	EN	AU	EU	AG	EG	AN	EN	AU	EU
`Ret`	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔
`Br`	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔
`Vis`	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔
$\emptyset$	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔
$\gg\!=$	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`iter`	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐

Fig. 14. Library of structural lemmas for `ICTree` combinators and `ticl` operators. Backwards-reasoning lemmas are indicated by ⇐ and bidirectional lemmas by ⇔.

$$\frac{\langle\ t,\ w \vDash_L\ \varphi\ \rangle}{\langle\ x \leftarrow t;; k\ x,\ w \vDash_L\ \varphi\ \rangle}\ \textsc{BindL}$$

$$\frac{\langle\ t \oplus u,\ w \vDash_L\ \varphi\ \rangle}{\langle\ t,\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle \quad \langle\ u,\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}\ \textsc{BrAU}_L \qquad \frac{\langle\ t \oplus u,\ w \vDash_L\ \varphi\ \rangle}{\langle\ t,\ w \vDash_L\ \varphi\ \mathsf{EU}\ \varphi'\ \rangle \vee \langle\ u,\ w \vDash_L\ \varphi\ \mathsf{EU}\ \varphi'\ \rangle}\ \textsc{BrEU}_L$$

$$\frac{\langle\ t,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ \mathcal{R}_Y\ \rangle \quad \forall\ y, w,\ \mathcal{R}_Y\ y\ w \to \langle\ k\ y,\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}{\langle\ x \leftarrow t;; k\ x,\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}\ \textsc{BindAU}_L \qquad \frac{\langle\ t,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}_{=}\ y\ w'\ \rangle \quad \langle\ k\ y,\ w' \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}{\langle\ x \leftarrow t;; k\ x,\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}\ \textsc{BindAU}_{L=}$$

$$\frac{\langle\ t,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ \mathcal{R}_Y\ \rangle \quad \forall\ y, w,\ \mathcal{R}_Y\ y\ w \to \langle\ k\ y,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \psi'_X\ \rangle}{\langle\ x \leftarrow t;; k\ x,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \psi'_X\ \rangle}\ \textsc{BindAU}_R \qquad \frac{\langle\ t,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ \mathcal{R}_Y\ \rangle \quad \forall\ y, w,\ \mathcal{R}_Y\ y\ w \to \langle\ k\ y,\ w \vDash_L\ \mathsf{AG}\ \varphi\ \rangle}{\langle\ x \leftarrow t;; k\ x,\ w \vDash_L\ \mathsf{AG}\ \varphi\ \rangle}\ \textsc{BindAG}$$

Fig. 15. Representative `ICTree` structural lemmas for nondeterminism and sequential composition.

next section (Section 6) we will see how to use `ticl` to define a new programming language, its denotation to `ICTrees` and its structural lemmas, with only a few lines of definitions. The table in Figure 14 shows the cartesian product of `ICTree` structures and `ticl` temporal operators. We have identified and proved backward-reasoning lemmas (⇐) for sequential composition and iteration ($\gg\!=$, `iter`) and bidirectional lemmas (⇔) for all `ICTree` nodes and nondeterministic choice ($\oplus$). We conjecture there are useful inversion lemmas for $\gg\!=$ and `iter` which we leave for future work.

The proof rules, collected in Figure 14, correspond to lemmas stated with regards to the entailment relation (Definition 1). All rules are proven—that is, each rule of our logic is sound—and the collection of all proofs corresponds to a statement of soundness for `ticl`. In our Rocq development a syntactic representation of entailment is provided, to facilitate automation. We prove that syntactic entailment implies semantic entailment by induction.

5.1 Sequential composition

In Section 3.1 we define the sequential composition of `ICTrees` (`Ret`,$\gg\!=$) and in Section 4.2 we motivate `ticl` postconditions (ψ_X), asserting the existence of compositional liveness lemmas. Figure 15 shows some of those lemmas. The goal is to distribute temporal specifications over the sequential ($\gg\!=$) and parallel ($\oplus$) composition of programs. As a result, we get modular subproofs for general liveness properties, analogous to the sequence rule for safety properties in Hoare logic.

For example, if $x \leftarrow t;; k\ x$ is a terminal application with the ability to print to standard output, and the goal is to prove that it will *eventually* print $\langle\ x \leftarrow t;; k\ x,\ w \vDash_L \texttt{AF obs PRINTS}\ \rangle$, there are two cases to consider:

(1) Either t prints to the terminal, use the BindL lemma (Figure 15) to prove it and ignore the continuation (k).
(2) Or the continuation k prints, use the BindAU$_L$ lemma to show t always terminates with postcondition $\mathcal{R}_Y$. Then for all possible return values ($y \in Y$) and worlds ($w' \in \mathcal{W}_E$) in the postcondition ($\mathcal{R}_Y\ y\ w'$), we must show the continuation ($k\ y$) eventually prints to the terminal $\langle\ k\ y,\ w' \vDash_L \texttt{AF obs PRINTS}\ \rangle$.

While structural lemmas are proven for both the universal (AN, AU, AG) and existential (EN, EU, EG) `ticl` operators, we focus our exposition on universal quantifiers, noting that the same lemmas apply to their existential versions.

For deterministic programs, the convenience lemma BindAU$_{L=}$ assumes that a linear path can be traversed in finite steps—remember the syntactic notation $\texttt{done}_=\ y\ w'$ introduced in Figure 10 uses equality to value y and world w' as the postcondition. This simplifying assumption lifts the need to manually specify postconditions of deterministic programs.[2]

5.2 Iteration

The iteration `ICTree` combinator (`iter`), defined in Section 3.1, encodes both finite and infinite loops. In this section we prove lemmas that show loop *termination*, *liveness*, and *invariance* (Figure 16) by using loop variant and invariant relations over the loop body.

The loop *termination* rule (IterAU$_R$) proves a loop terminates with postcondition ψ_X. It requires specifying a loop invariant relation ($\mathcal{R}$), and a binary *well-founded* relation called the loop variant (R_v)—well-founded relations have no infinite chains, ensuring the loop terminates in finite steps. There are two obligations—corresponding to the inductive step, and the base case of the underlying induction.

(1) If the loop body's return value $lr = \texttt{inl}\ i'$, the loop continues. The loop invariant $\mathcal{R}$ must be satisfied before and after the loop body, much like in Hoare Logic. The next iteration (represented by iterator i' and world w') must be "smaller" according to the loop variant ($\mathcal{R}_v\ (i', w')\ (i, w)$).
(2) If $lr = \texttt{inr}\ r$ the loop terminates concluding the proof. All that is left is to show that the loop returns a state satisfying the loop postcondition $\langle\ \texttt{Ret}\ r,\ w \vDash_R \varphi\ \texttt{AN}\ \psi_X\ \rangle$.

The loop *liveness* rule (IterAU$_L$) is slightly different than the termination rule: it expects formula φ' to be eventually satisfied, even if the loop keeps running afterwards—possibly forever. Note that this is the only meaning of liveness in temporal logics like LTL and CTL. `Ticl` differentiates between those two very different cases with the two rules (IterAU$_R$ and IterAU$_L$) in Figure 16—the differences are further highlighted in Section 8.2. Similar to loop *termination*, the *liveness rule* expects two relations—the loop invariant ($\mathcal{R}$) and the loop variant ($\mathcal{R}_v$)—and produces two proof obligations, except now we get to chose which one is satisfied in each iteration ($k\ i$)

(1) Either the loop body satisfies the liveness property $\langle\ \texttt{k i},\ w \vDash_L \varphi\ \texttt{AU}\ \varphi'\ \rangle$.
(2) Or, the loop continues ($lr = \texttt{inl}\ i'$) and the new iterator (i') and world (w') satisfy the invariant ($\mathcal{R}$) and are "smaller" with respect to the well-founded variant ($\mathcal{R}_v$).

Working directly with well-founded relations in Rocq can be difficult, so we define simplified versions of rules IterAU$_R$ and IterAU$_L$ expecting a *ranking function* (IterAU$_{L,\mathbb{N}}$). A ranking function

[2]In practice, y and w' are replaced with existential variables in Rocq proofs, delaying their instantiation until program t returns, largely automating the use of this rule.

$$\frac{\begin{array}{l}\mathcal{R}\ i\ w \qquad \texttt{well\_founded}\ \mathcal{R}_v \\ \forall\ i, w, \mathcal{R}\ i\ w \to \\ \quad \langle\ \texttt{k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle \vee \\ \quad \langle\ \texttt{k i},\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \text{done}\ (\lambda\ lr\ w' \Rightarrow \\ \qquad \exists\ i',\ lr = \texttt{inl}\ i' \wedge \mathcal{R}\ i'\ w' \\ \qquad \wedge\ \mathcal{R}_v\ (i', w')\ (i, w))\ \rangle\end{array}}{\langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}\ \textsc{IterAU}_L$$

$$\frac{\begin{array}{l}\mathcal{R}\ i\ w \\ \forall\ i,\ w,\ \mathcal{R}\ i\ w \to \\ \quad \langle\ \texttt{k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle \vee \\ \quad \langle\ \texttt{k i},\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \text{done}\ (\lambda\ lr\ w' \Rightarrow \\ \qquad \exists\ i',\ lr = \texttt{inl}\ i' \wedge \mathcal{R}\ i'\ w' \\ \qquad \wedge\ f\ i'\ w' < f\ i\ w)\ \rangle\end{array}}{\langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}\ \textsc{IterAU}_{L,\mathbb{N}}$$

$$\frac{\begin{array}{l}\mathcal{R}\ i\ w \qquad \texttt{well\_founded}\ \mathcal{R}_v \\ \forall\ i,\ w,\ \mathcal{R}\ i\ w \to \\ \quad \langle\ k\ i,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \text{done}\ (\lambda\ lr\ w' \Rightarrow \\ \qquad \begin{cases} \mathcal{R}\ i'\ w' \wedge \mathcal{R}_v\ (i', w')\ (i, w), & \text{if lr} = \texttt{inl}\ i' \\ \langle\ \mathsf{Ret}\ r,\ w' \vDash_R\ \varphi\ \mathsf{AN}\ \psi_X\ \rangle, & \text{if lr} = \texttt{inr}\ r \end{cases} \\ )\ \rangle\end{array}}{\langle\ \texttt{iter k i},\ w \vDash_R\ \varphi\ \mathsf{AU}\ \psi_X\ \rangle}\ \textsc{IterAU}_R$$

$$\frac{\begin{array}{l}\mathcal{R}\ i\ w \\ \forall\ i,\ w,\ \mathcal{R}\ i\ w \to \\ \quad \langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \rangle \wedge \\ \quad \langle\ \texttt{k i},\ w \vDash_R\ \mathsf{AX}(\varphi\ \mathsf{AU}\ \mathsf{AX}\ \text{done}\ (\lambda\ lr\ w' \Rightarrow \\ \qquad \exists\ i',\ lr = \texttt{inl}\ i' \wedge \mathcal{R}\ i'\ w'))\ \rangle\end{array}}{\langle\ \texttt{iter k i},\ w \vDash_L\ \mathsf{AG}\ \varphi\ \rangle}\ \textsc{IterAG}$$

Fig. 16. Representative `ticl` iteration lemmas for operators AU and AG and ICTrees.

maps iterators and worlds to the natural numbers ($f \in I \to \mathcal{W}_E \to \mathbb{N}$), such that successive pairs of iterator and world are strictly monotonically decreasing. Finding suitable ranking functions for complex loops can be challenging. In Example 7.3 we demonstrate a new lemma we call *liveness split*, that reduces liveness proofs to smaller liveness proofs, with smaller ranking functions. Recent, orthogonal work on automatic inference of ranking functions [51] also works well with `ticl`, as `ticl`'s iteration rules can use such inferred ranking functions to produce formal proofs of liveness.

`Ticl` addresses *nonterminating* loops with the *invariance* rule (IterAG) in Figure 16. This rule can prove both *always* and *always-eventually* properties by specifying a suitable loop invariant $\mathcal{R}$ and two proof obligations:

(1) The inner formula φ must hold for the whole loop $\langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \rangle$. For example, if φ is an *eventually* property the loop might run multiple times before satisfying the base formula.
(2) The loop body ($k\ i$) must take at least one step and eventually terminate, satisfying φ in every step. Stepping once is required for the loop to be productive, preventing unsoundness issues with cyclic proofs [38]. In addition, the loop body ($k\ i$) must always continue ($lr = \texttt{inl}\ i'$), and the new iterator i' and world w' must satisfy the loop invariant ($\mathcal{R}$).[3]

The *invariance* rule (IterAG) is significant because it discharges a coinductive (infinite) proof to two finite subproofs. This single rule encapsulates all of `ticl`'s coinduction techniques, and it has demonstrated to be sufficiently general to complete every *always* and *always-eventually* proof in our evaluation.

[3]It is not always the case the loop body will terminate. For example, there may be nested infinite loops. Unfolding the outer `iter` loop reveals its definition in terms of bind (≫=). The BindL rule in Figure 15 eliminates the outer loop, allowing us to apply the *invariance* rule on the inner loop.

$$
\begin{array}{lll}
\texttt{AExp} \in \mathit{Type} = & \mid \texttt{var}\ (s \in \mathit{string}) & \mid \texttt{val}\ (n \in \mathbb{N}) \\
& \mid (x \in \texttt{AExp})\ \texttt{+}\ (y \in \texttt{AExp}) & \mid (x \in \texttt{AExp})\ \texttt{-}\ (y \in \texttt{AExp}) \\
\texttt{BExp} \in \mathit{Type} = & \mid (x \in \texttt{AExp})\ \texttt{<}\ (y \in \texttt{AExp}) & \\
\texttt{StImp} \in \mathit{Type} = & \mid (s \in \mathit{string}) \leftarrow (y \in \texttt{AExp}) & \mid \textbf{if}\ (c \in \texttt{BExp})\ \textbf{then}\ (x \in \texttt{StImp})\ \textbf{else}\ (y \in \texttt{StImp}) \\
& \mid (l \in \texttt{StImp})\ \texttt{;}\ (r \in \texttt{StImp}) & \mid \textbf{while}\ (c \in \texttt{BExp})\ \{t \in \texttt{StImp}\} \quad \mid \texttt{skip}
\end{array}
$$

Fig. 17. Syntax of a small imperative language StImp with mutable state and nondeterminism.

$$
\begin{array}{l}
\texttt{state}_{\mathcal{M}} \in \texttt{Type} \to \texttt{Type} = \quad \mid (\texttt{Get} \in \texttt{state}_{\mathcal{M},\mathcal{M}}) \quad \mid (\texttt{Put}\ (\texttt{m} \in \mathcal{M}) \in \texttt{state}_{\mathcal{M},\texttt{unit}}) \\
\texttt{get} \in \texttt{ictree}_{\texttt{state}_{\mathcal{M}},\,\mathcal{M}} = \quad \texttt{trigger Get} \\
\texttt{put}\ (m \in \mathcal{M}) \in \texttt{ictree}_{\texttt{state}_{\mathcal{M}},\,\texttt{unit}} = \quad \texttt{trigger}\ (\texttt{Put m}) \\
\\
\texttt{h}_{\mathcal{M}} \in \texttt{state}_{\mathcal{M}} \rightsquigarrow \texttt{InstrM}_{\mathcal{M},\mathcal{M}} \\
h_{\mathcal{M}}\ (\texttt{Get} \in \texttt{state}_{\mathcal{M},\mathcal{M}})\ (m \in \mathcal{M}) = \texttt{Ret}\ (m, m) \\
h_{\mathcal{M}}\ (\texttt{Put m}' \in \texttt{state}_{\mathcal{M},\texttt{unit}})\ (\_ \in \mathcal{M}) = \log m' \,;;\, \texttt{Ret}\ ((), m') \\
[\![\_]\!]_{\texttt{A}} \in \texttt{AExp} \to \texttt{ictree}_{\texttt{state}_{\mathcal{M}},\,\mathbb{N}} \\
[\![\texttt{var}\ s]\!]_{\texttt{A}} = \texttt{get} \gg\!= (\lambda\ m \Rightarrow \texttt{Ret}\ m[s]), \qquad [\![x\ \texttt{+}\ y]\!]_{\texttt{A}} = a \leftarrow [\![x]\!]_{\texttt{A}} \,;;\, b \leftarrow [\![y]\!]_{\texttt{A}} \,;;\, \texttt{Ret}\ (a + b) \\
[\![\texttt{val}\ n]\!]_{\texttt{A}} = \texttt{Ret}\ n, \qquad\qquad\qquad [\![x\ \texttt{-}\ y]\!]_{\texttt{A}} = a \leftarrow [\![x]\!]_{\texttt{A}} \,;;\, b \leftarrow [\![y]\!]_{\texttt{A}} \,;;\, \texttt{Ret}\ (a - b) \\
[\![\_]\!]_{\texttt{B}} \in \texttt{BExp} \to \texttt{ictree}_{\texttt{state}_{\mathcal{M}},\,\mathbb{B}} \\
[\![x\ \texttt{<}\ y]\!]_{\texttt{B}} = a \leftarrow [\![x]\!]_{\texttt{B}} \,;;\, b \leftarrow [\![y]\!]_{\texttt{B}} \,;;\, \texttt{Ret}\ (a < b) \\
[\![\_]\!]_{\texttt{S}} \in \texttt{StImp} \to \texttt{ictree}_{\texttt{state}_{\mathcal{M}},\,\texttt{unit}} \\
[\![s \leftarrow x]\!]_{\texttt{S}} = a \leftarrow [\![x]\!]_{\texttt{S}} \,;;\, m \leftarrow \texttt{get} \,;;\, \texttt{put}\ ((s \hookrightarrow a) \cup m), \quad [\![t\ \texttt{;}\ u]\!]_{\texttt{S}} = [\![t]\!]_{\texttt{S}} \,;;\, [\![u]\!]_{\texttt{S}} \\
[\![\textbf{if}\ (c)\ \textbf{then}\ t\ \textbf{else}\ u]\!]_{\texttt{S}} = [\![c]\!]_{\texttt{B}} \gg\!= \left(\lambda\ (cv \in \mathbb{B}) \Rightarrow \begin{cases} [\![t]\!]_{\texttt{S}}, & \text{if } cv \\ [\![u]\!]_{\texttt{S}}, & \text{otherwise} \end{cases}\right), \quad [\![\texttt{skip}]\!]_{\texttt{S}} = \texttt{Ret}\ (), \\
[\![\textbf{while}\ (c)\ \{t\}]\!]_{\texttt{S}} = \texttt{iter}\ \left(\lambda\ () \Rightarrow [\![\texttt{c}]\!]_{\texttt{B}} \gg\!= \left(\lambda\ \texttt{cv} \Rightarrow \begin{cases} [\![t]\!]_{\texttt{S}} \,;;\, \texttt{Ret}\ (\texttt{inl}\ ()), & \text{if } cv \\ \texttt{Ret}\ (\texttt{inr}\ ()), & \text{otherwise} \end{cases}\right)\right)\ () \\
\\
[(x \in \texttt{AExp}),\ (m \in \mathcal{M}) \Vdash_{LR}\ p]_A \in \mathbb{P} = \quad \langle\ \texttt{instr}\ h_{\mathcal{M}}\ [\![x]\!]_{\texttt{A}}\ m,\ \texttt{Obs}\ (\texttt{Log}\ m)\ () \vDash_{LR}\ p\ \rangle \\
[(c \in \texttt{BExp}),\ (m \in \mathcal{M}) \Vdash_{LR}\ p]_B \in \mathbb{P} = \quad \langle\ \texttt{instr}\ h_{\mathcal{M}}\ [\![c]\!]_{\texttt{B}}\ m,\ \texttt{Obs}\ (\texttt{Log}\ m)\ () \vDash_{LR}\ p\ \rangle \\
[(t \in \texttt{StImp}),\ (m \in \mathcal{M}) \Vdash_{LR}\ p]_S \in \mathbb{P} = \quad \langle\ \texttt{instr}\ h_{\mathcal{M}}\ [\![t]\!]_{\texttt{S}}\ m,\ \texttt{Obs}\ (\texttt{Log}\ m)\ () \vDash_{LR}\ \varphi\ \rangle
\end{array}
$$

Fig. 18. Instrumentation and entailment of StImp programs by denotation to $\texttt{ictree}_{\texttt{state}_{\mathcal{M}}}$.

6 Using Ticl and ICTrees

This section demonstrates how to use ticl and ICTrees to define new programming languages and temporal proof systems. Starting with a simple imperative language called StImp, we give a step-by-step recipe to get started with ticl liveness proofs:

(1) Define the syntax and denotational semantics of StImp using ICTrees (Figures 17 and 18).
(2) Define an instrumentation handler from StImp events ($\texttt{state}_{\mathcal{M}}$) to an instrumentation monad ($\texttt{InstrM}_{\mathcal{M},\mathcal{M}}$), chosing an appropriate ghost-state to observe ($h_{\mathcal{M}}$ in Figure 18).
(3) Using the ticl library of structural lemmas (Section 5), prove high-level structural rules for StImp (Figure 19).

$$\frac{[c,\ m \Vdash_R\ \mathsf{AX}\ \mathsf{done}_=\ (b,m)]_B \quad \wedge \begin{cases} [t,\ m \Vdash_R\ \varphi\ \mathsf{AU}\ \varphi']_S, & \text{if } b \\ [u,\ m \Vdash_R\ \varphi\ \mathsf{AU}\ \varphi']_S, & \text{otherwise} \end{cases}}{[\textbf{if}\ (c)\ \textbf{then}\ t\ \textbf{else}\ u,\ m \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_S}\ \textsc{If}_S\mathsf{AU}_L$$

$$\frac{[t,\ m \Vdash_R\ \varphi\ \mathsf{AU}\ \mathsf{done}_=\ m']_S \quad [u,\ m' \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_S}{[t\ ;\ u,\ m \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_S}\ \textsc{Seq}_S\mathsf{AU}_L$$

$$\frac{\mathcal{R}\ m \to \forall\ m,\ \mathcal{R}\ m \to [c,\ m \Vdash_R\ \mathsf{AX}\ \mathsf{done}_=\ (b,m)]_B \quad \wedge \begin{cases} [t,\ m \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_S \vee \\ [t,\ m \Vdash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ (\lambda\ m' \Rightarrow \\ \quad \mathcal{R}\ m'\ \wedge\ f\ m' < f\ m)\ ]_S, & \text{if } b \\ [\mathtt{skip},\ m \Vdash_L\ \varphi']_S, & \text{otherwise} \end{cases}}{[\textbf{while}\ (c)\ \{t\},\ m \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_S}\ \textsc{While}_S\mathsf{AU}_L$$

$$\frac{\begin{array}{l}\mathcal{R}\ m \to \forall\ m,\ \mathcal{R}\ m \to \\ \quad [\textbf{while}\ (c)\ \{t\},\ m \Vdash_L\ \varphi]_S \wedge \\ \quad [c,\ m \Vdash_R\ \mathsf{AX}\ \mathsf{done}_=\ (\mathtt{true},m)]_B \wedge \\ \quad [t,\ m \Vdash_R\ \mathsf{AX}(\varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ \mathcal{R})]\end{array}}{[\textbf{while}\ (c)\ \{t\},\ m \Vdash_L\ \mathsf{AG}\ \varphi]_S}\ \textsc{While}_S\mathsf{AG}$$

Fig. 19. Representative structural lemmas for language StImp and ticl operators AU, AG.

Ticl works at a high-level of abstraction—all of the above fits in 500 lines of Rocq proofs and definitions—a small number even compared to program logics for safety properties. Steps (1) and (2) above are the same as those required for working with Interaction Trees [50]. The proofs of the language-specific structural rules in step (3) are high-level, syntactic, and fairly repetitive.

6.1 Instrumentation of StImp

In Section 3.2 we defined *instrumentation*, a mechanism to evaluate events and record proof relevant ghost-state. In this section, we demonstrate the process of giving semantics to StImp programs in two stages: (1) denoting StImp syntax to an ICTree, and (2) instrumenting the ICTree to an appropriate instrumentation monad.

For the first stage, the denotation brackets ($[\![\_]\!]_s$) in Figure 18 translate the syntax of StImp (Figure 17) to an $\mathtt{ictree}_{\mathtt{state}_{\mathcal{M}}}$. The shared state $\mathcal{M}$ is a map from string indices to natural number ($\mathbb{N}$) values. Low-level operations on maps and their lemmas are assumed; $m_1 \cup m_2$ is map union, $s \hookrightarrow x$ is the singleton map with key s and value x, $m[s]$ is the partial "get" that returns an option $\mathtt{Some}(v)$, such that v is the value associated with key s, or None if key s does not exist in m. An StImp var s expression retrieves the value of variable s from shared state ($m[s]$), while an assignment ($s \leftarrow x$) statement updates the shared memory with the new value ($(s \hookrightarrow a) \cup m$).

The second stage defines the StImp instrumentation handler ($h_{\mathcal{M}}$) in Figure 18. This handler gives a semantic meaning to get and put events, while also specifying the *ghost-state* to be observed by the proof system. We chose to only instrument put m events—by calling log m—to ensure that the instrumentation monad only remembers events *overwriting* the state, while events reading from the state are evaluated and erased. Our choice of handler affects what properties can be proved in ticl later. For example, we cannot prove "an empty heap is read" using a handler that erases reads ($h_{\mathcal{M}}$). Handlers are a flexible mechanism for observing program behavior; we will see more examples of that in Section 7.

Putting the two stages together, instr $h_{\mathcal{M}}$ "applies" the instrumentation handler over the tree ($[\![t]\!]_s \in \mathtt{ictree}_{\mathtt{state}_{\mathcal{M}}}$) with initial state s. The end-to-end entailment relation $[t,\ s \Vdash_{LR}\ p]_S$, connects an StImp program (t) to a ticl specification (p).

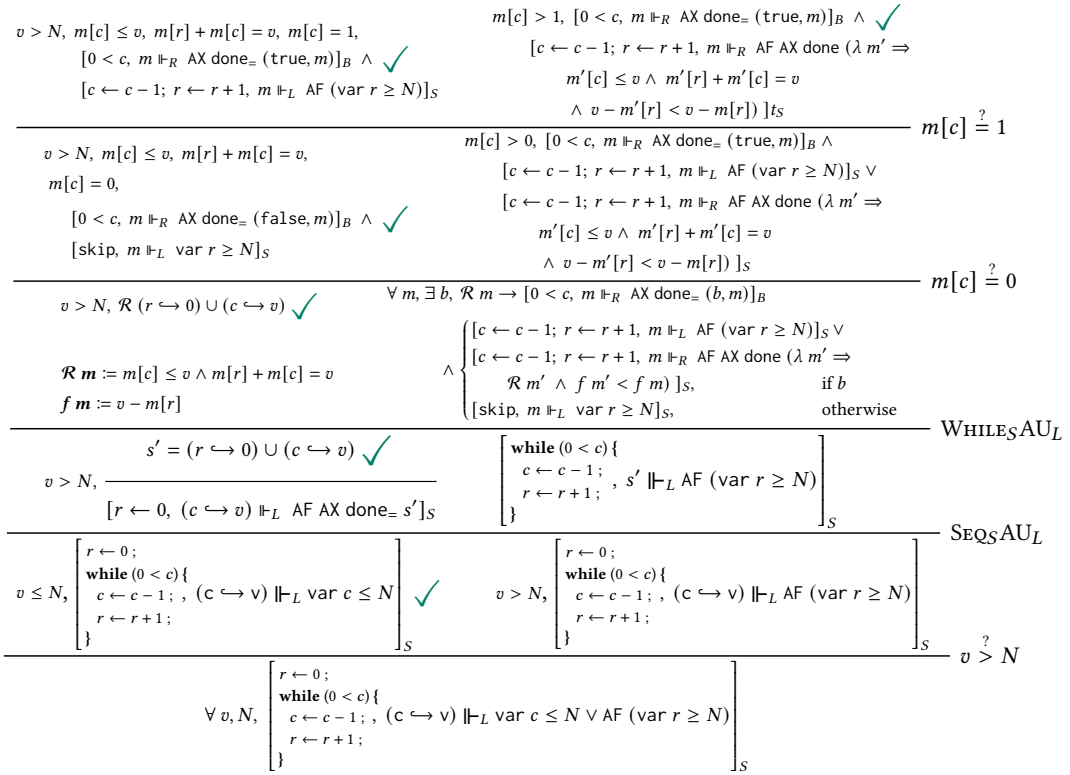

Fig. 20. Example structural liveness proof for a simple StImp loop program using ticl.

6.2 StImp structural lemmas

Equipped with the end-to-end ticl entailments over StImp programs, we proceed to "lift" the ICTree rules in Figures 15 and 16 to the level of StImp program structures. Representative StImp lemmas—focusing on operators AU and AG—are shown in Figure 19. The full array of program structures and temporal opertors is proven in our development.

Structural rules for StImp (Figure 19)—much like structural rules of ICTrees—are backwards reasoning, meaning the goal is in the bottom and proof obligations are given on the top of the inference line. The obligations generated are "smaller" than the goal they apply to, either targeting a subprogram of the original program, or a subformula of the original formula. For example, in the *invariance* rule WhileAG, the first proof obligation φ is a subformula of AG φ, while the other two proof obligations refer to the loop conditinal and loop body.

6.3 Example: structural proof of liveness for StImp

Now we demonstrate a ticl structural proof in practice, by proving liveness of a program from the T2 CTL benchmark suite [6]. The program in Figure 20 (bottom) is a simple **while** loop. The goal specification has been generalized with quantifiers ($\forall\, v, N$) which are known to be challenging for model checking systems [6, 13, 21, 37], but not Ticl and the Rocq proof assistant. We want to prove that either the initial value of variable c is $\le$ a constant ($N \in \mathbb{N}$), or eventually the value of r

becomes $\geq$ to N. Base formulas (for example $\texttt{var}\ n > 0$) dereference variable n from the current state (notation for $\exists\, v, m[n] = \texttt{Some}(v)$ and $v > 0$).

The proof begins by case analysis on value v. If $v \leq N$, we have proved the left-side of the "or" in our goal. Here, v is the initial value of variable c. Otherwise if $v > N$, use the sequence rule ($\textsc{Seq}_S\textsc{AU}_L$ in Figure 19) to update the state (s') with the new assignment ($r \leftarrow 0$).

At this point, the **while** loop starts at the new state (s') and our goal is to prove the "eventually" property $\texttt{AF}\ (\texttt{var}\ r \geq N)$. The liveness lemma ($\textsc{While}_S\textsc{AU}_L$ in Figure 19) applies. For its loop invariant ($\mathcal{R}$), notice the sum of values in r and c remains constant throughout the loop ($m[r] + m[c] = v$)[4]. For its ranking function (f), notice that variable c is at its greatest at value v, progressively decreasing every iteration ($f\ m \coloneqq v - m[c]$). The remaining proof is straightforward.

(1) Prove the initial loop state satisfies the loop invariant ($\mathcal{R}\ (r \hookrightarrow 0) \cup (c \hookrightarrow v)$).
(2) For each state (m) satisfying the invariant $\mathcal{R}\ m$, the loop body must terminate and satisfy the loop invariant and the ranking function condition.
(3) Taking three cases on the value of c concludes the proof in Figure 20:
    - If $m[c] = 0$ then $m[r] = v$, so $\texttt{var}\ r \geq N$ is true.
    - If $m[c] = 1$ then $m[r] = v - 1$, the loop body adds 1 to r, so again, $\texttt{var}\ r \geq N$.
    - If $m[c] > 1$ then it is easy to prove the loop invariant and variant are satisfied at termination of the loop body.

7 Motivating examples

We evaluated `ticl` by structurally verifying several examples from the T2 CTL benchmark suite [6] (like the example in Figure 20) and three use cases inspired from computer systems. In this section, we prove liveness and safety properties for a round-robin scheduler, a secure concurrent shared memory system, and a distributed consensus protocol.

In Section 6 we defined the deep-embedding[5] imperative language `StImp` and its `ticl` theory. In this section we define three mixed embedding languages (`MeQ`, `MeS`, `MeR`) [12] and a shallow-embedding language ($\texttt{ictree}_{E_{\text{net}}}$). Our goal is to demonstrate the flexibility of `ticl` over different programming language techniques, events, and temporal specifications.

7.1 Round-robin scheduler

The syntax of the `MeQ` language for the round-robin scheduler (`rr`) from Figure 1 is given in Figure 21. A shared queue (Q) with elements T maintains the order of threads in the scheduler. Language `MeQ` interfaces with the queue through the `pop` and `push` instructions and has sequential composition ($\gg\!=$) and an infinite loop program structures (**while** (`true`) $\{t\}$). This simple language is sufficient to prove that a thread x will *always-eventually* get scheduled, the nested temporal property we saw early on in Figure 1. Nested temporal properties like always-eventually pose a challenge for deductive verification, not only to prove but to formally state.

Following the steps in Section 6, the denotation of `MeQ` programs ($\texttt{ictree}_{E_Q}$) is shown in Figure 22. The queue instrumentation handler (h_Q) keeps track of popped elements, but not pushed elements. The scheduler performs both actions in sequence, so provenance information is not lost through this choice; still, different target properties might require defining a different handler.

[4]Existantials are omitted in natural number propositions, so $m[r] + m[c] = v$ is shorthand notation for $\exists\, v_r,\ m[r] = \texttt{Some}(v_r)$ and $\exists\, v_c,\ m[c] = \texttt{Some}(v_c)$ such that $v_r + v_c = v$.

[5]Deep-embedding refers to the treatment of variables as datatypes and the explicit handling of substitution. Shallow-embedding languages avoid using a datatype for their abstract syntax altogether, in favor of programming in the proof assistant's metalanguage. Mixed-embedding languages have an abstract syntax, but use the proof assistant's variable and substitution mechanisms [12].

$$
\begin{array}{l}
\mathcal{Q} \in \mathtt{Type} = \ \mathtt{list}\ T \\
E_{\mathcal{Q}} \in \mathtt{Type} \to \mathtt{Type} = \ \mid \mathsf{Push}\ (x \in T) \in E_{\mathcal{Q}}\ \mathtt{unit} \ \mid \mathsf{Pop} \in E_{\mathcal{Q}}\ T \\
\mathtt{MeQ} \in \mathtt{Type} \to \mathtt{Type} = \ \mid \mathsf{pop} \in \mathtt{MeQ}_T \ \mid \mathsf{push}\ (x \in T) \in \mathtt{MeQ}_{\mathtt{unit}} \\
\qquad\qquad\qquad\quad \mid \mathbf{while}\ (\mathtt{true})\ \{p \in \mathtt{MeQ}_X\} \in \mathtt{MeQ}_{\mathtt{unit}} \\
\qquad\qquad\qquad\quad \mid (a \in \mathtt{MeQ}_A) \gg\!= (k \in A \to \mathtt{MeQ}_B) \ \mid \mathtt{ret}\ (a \in A) \in \mathtt{MeQ}_A
\end{array}
$$

Fig. 21. Language MeQ for a round-robin scheduler with a mutable queue ($\mathcal{Q}$).

$$
\begin{array}{l}
\mathsf{h}_{\mathcal{Q}} \in E_{\mathcal{Q}} \rightsquigarrow \mathtt{InstrM}_{\mathcal{Q},T} \\
h_{\mathcal{Q}}\ (\mathsf{Push}\ n \in E_{\mathcal{Q}}\ \mathtt{unit})\ (q \in \mathcal{Q}) \ = \ \mathtt{Ret}\ ((), q \mathbin{+\!\!+} [n]) \\
h_{\mathcal{Q}}\ (\mathsf{Pop} \in E_{\mathcal{Q}}\ \mathbb{N})\ (h{::}ts \in \mathcal{Q}) \ = \ \mathtt{log}\ h \mathbin{;;} \mathtt{Ret}\ (h, ts) \\
h_{\mathcal{Q}}\ (\mathsf{Pop} \in E_{\mathcal{Q}}\ \mathbb{N})\ ([] \in \mathcal{Q}) \ = \ \emptyset \\
[\![\_]\!]_{\mathcal{Q}} \in \mathtt{MeQ}_A \to \mathtt{ictree}_{E_{\mathcal{Q}},\,A} \\
[\![\mathsf{pop}]\!]_{\mathcal{Q}} \ = \ \mathtt{trigger}\ \mathsf{Pop}, \quad [\![\mathsf{push}\ x]\!]_{\mathcal{Q}} \ = \ \mathtt{trigger}\ (\mathsf{Push}\ x), \\
[\![\mathbf{while}\ (\mathtt{true})\ \{t\}]\!]_{\mathcal{Q}} \ = \ \mathtt{iter}\ \left(\lambda\ () \Rightarrow [\![\mathtt{t}]\!]_{\mathcal{Q}} \gg\!= (\lambda\ \_ \Rightarrow \mathtt{Ret}\ (\mathtt{inl}\ ()))\right)\ (), \\
[\![x \gg\!= k]\!]_{\mathcal{Q}} \ = \ [\![x]\!]_{\mathcal{Q}} \gg\!= (\lambda\ a \Rightarrow [\![k\ a]\!]_{\mathcal{Q}}), \quad [\![\mathtt{ret}\ x]\!]_{\mathcal{Q}} \ = \ \mathtt{Ret}\ x, \\
[(t \in \mathtt{MeQ}_A),\ (q \in \mathcal{Q}),\ (w \in \mathcal{W}_{\mathtt{L}_T}) \Vdash_{LR}\ p]_{\mathcal{Q}} \in \mathbb{P} \ = \ \ \langle\ \mathtt{instr}\ h_{\mathcal{Q}}\ [\![t]\!]_{\mathcal{Q}}\ q,\ w \vDash_{LR}\ p\ \rangle
\end{array}
$$

Fig. 22. Denotation $[\![\_]\!]_{\mathcal{Q}}$ and `ticl` entailment for queue language MeQ.

$$
\frac{\begin{array}{l}\mathcal{R}\ q\ w \quad \to \forall\ q\ w,\ \mathcal{R}\ q\ w \to \\ \quad [t,\ q,\ w \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_{\mathcal{Q}} \vee \\ \quad [t,\ q,\ w \Vdash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ (\lambda\ q'\ w' \Rightarrow \\ \qquad \mathcal{R}\ q'\ w' \ \wedge\ f\ q' < f\ q)\ ]_{\mathcal{Q}}\end{array}}{[\mathbf{while}\ (\mathtt{true})\ \{t\},\ q,\ w \Vdash_L\ \varphi\ \mathsf{AU}\ \varphi']_{\mathcal{Q}}}\ \textsc{While}_{\mathcal{Q}}\mathrm{AU}_L
\qquad
\frac{\begin{array}{l}\mathcal{R}\ q\ w \to \forall\ q\ w,\ \mathcal{R}\ q\ w \to \\ \quad [\mathbf{while}\ (\mathtt{true})\ \{t\},\ q,\ w \Vdash_L\ \varphi]_{\mathcal{Q}} \wedge \\ \quad [t,\ q,\ w \Vdash_R\ \mathsf{AX}\ (\varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ \mathcal{R})\ ]_{\mathcal{Q}}\end{array}}{[\mathbf{while}\ (\mathtt{true})\ \{t\},\ q,\ w \Vdash_L\ \mathsf{AG}\ \varphi]_{\mathcal{Q}}}\ \textsc{While}_{\mathcal{Q}}\mathrm{AG}
$$

Fig. 23. Liveness and invariance loop lemmas for queue language MeQ.

The end-to-end `ticl` entailment relation for MeQ $[t,\ q,\ w \Vdash_{LR}\ p]_{\mathcal{Q}}$ is a *quaternary* relation—in contrast to previous ternary entailments we have seen. Its arguments are the program t, queue q, current world w, and `ticl` formula p. In the last example (Figure 18) we used a ternary relation, because the ghost-state coincided with the program state ($\mathcal{M}$). However, in Figure 22 the ghost-state represents elements popped (T) while the program state represents the queue ($\mathcal{Q}$), which necessitates keeping track of both.

At this point we can prove the *liveness* and *invariance* lemmas in Figure 23, which we will need to complete the *always-eventually* proof in Figure 24. These proofs are short and reuse the `iter` lemmas from Figure 16. Comparing with the StImp loop lemmas in Figure 19, the MeQ loop lemmas are simpler; MeQ only supports infinite loops so there is no case analysis on the loop conditional.

Finally, we proceed backwards (from bottom to top) through the always-eventually proof in Figure 24. Start by using the *invariance* rule $\textsc{While}_{\mathcal{Q}}\mathrm{AG}$ with loop invariant ($\mathcal{R}$): world w must be not done, the queue will never be empty ($\exists\ h, ts,\ q = h :: ts$), and either the head element is the target thread ($h = x$), or not ($h \neq x$), in which case x must appear at some position i in the queue's tail ($\mathtt{find}\ x\ ts = \mathtt{Some}(i)$). Applying the invariance rule leaves three proof obligations:

(1) The loop invariant must be initially satisfied ($\mathcal{R}\ (q \mathbin{+\!\!+} [x])\ w$).

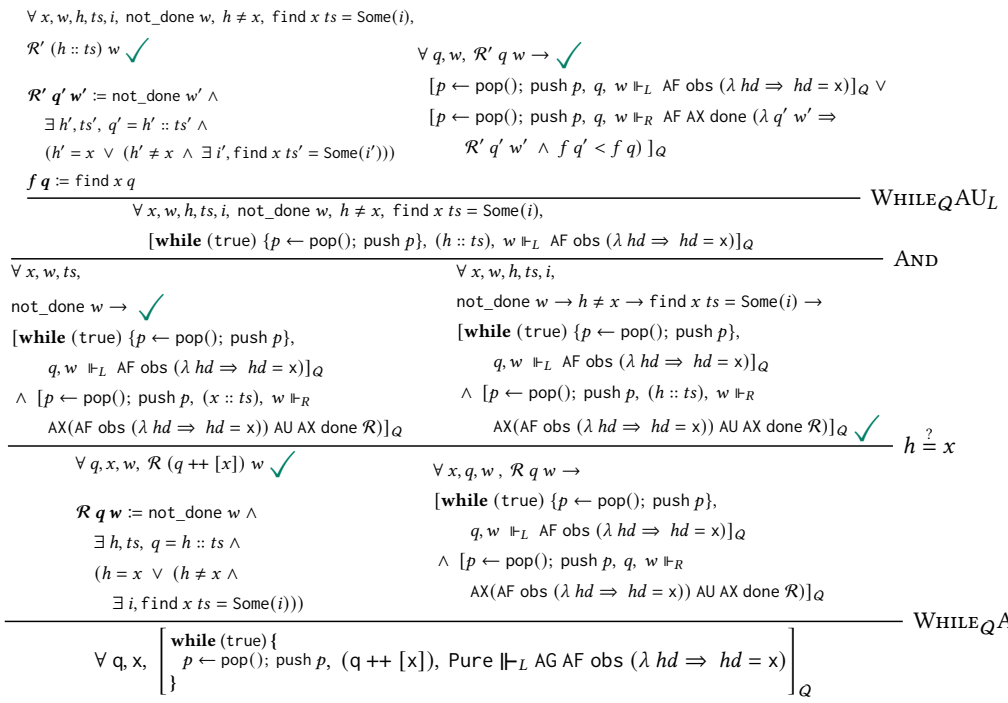

Fig. 24. Structural always-eventually proof for round-robin in MeQ using `ticl`.

(2) The loop body steps, then eventually terminates, respecting the invariant $\mathcal{R}$.
(3) The loop must satisfy the inner *eventually* property (AF obs ($\lambda\ hd \Rightarrow\ hd$ = x)).

The first subproof is easy to prove. The second is also straightforward by case analysis on the head of the queue ($h = x$). The third subproof (inner eventually) requires the *liveness* lemma (WHILE$_Q$AU$_L$) from Figure 23. Since this is the same loop as before, we can reuse the loop invariant ($\mathcal{R}$) from the *invariance* rule. The ranking function f is given simply by the index of x in the queue. The queue always contains x (by $\mathcal{R}$), so we can treat f as total. We conclude the proof by low-level reasoning over lists and using the loop invariants. For further details the reader can refer to our Rocq development; the syntax and semantics of MeQ, (Figure 21), `ticl` structural lemmas (Figure 23) and the *always-eventually* proof in Figure 24 required a total of 137 lines of Rocq definitions and 362 lines of proofs.

7.2 Secure concurrent shared memory

For the next example let us switch gears and prove confidentiality of concurrent reads and writes over a shared memory with security labels—a safety property. The proposed system is inspired by Mandatory Access Control (MAC). Language MeS in Figure 25 uses a mutable heap ($\mathcal{M}_S$), where each cell is *tagged* with an information-flow security label, either *low* security (L) or high security (H). Labels form a preorder with accessibility relation ($\leq$)—the smallest reflexive, transitive relation such that $L \leq H$ holds.

Tagged memory is accessed by instructions `read` $l_i\ x$ and `write` $l_i\ x\ y$, where l_i is the permission level of the instruction, x is the address, and y is the value to write. The goal is to prove every read instruction (`read` $l_i\ x$) accesses a memory cell with a security level that is $\leq$ to its permission level

$$
\begin{array}{lll}
\mathcal{S} \in \texttt{Type} = & \mid L \quad \mid H & \\
\mathcal{M}_{\mathcal{S}} \in \texttt{Type} = & \texttt{Map}_{\mathbb{N},(\mathbb{N} * \mathcal{S})} & \\
E_{\mathcal{S}} \in \texttt{Type} \to \texttt{Type} = & \mid \texttt{Read}\ (l \in \mathcal{S})\ (x \in \mathbb{N}) \in E_{\mathcal{S}}\ \texttt{option}_{\mathbb{N}} & \\
& \mid \texttt{Write}\ (l \in \mathcal{S})\ (x \in \mathbb{N})\ (v \in \mathbb{N}) \in E_{\mathcal{S}}\ \texttt{unit} & \\
\texttt{MeS} \in \texttt{Type} \to \texttt{Type} = & \mid \texttt{read}\ (l \in \mathcal{S})\ (n \in \mathbb{N}) \in \texttt{MeS}_{\texttt{option}_{\mathbb{N}}} & \\
& \mid \texttt{write}\ (l \in \mathcal{S})\ (n \in \mathbb{N})\ (v \in \mathbb{N}) \in \texttt{MeS}_{()} & \\
& \mid \textbf{if}\ (c \in \mathbb{B})\ \textbf{then}\ (t \in \texttt{MeS}_A)\ \textbf{else}\ (u \in \texttt{MeS}_A) \in \texttt{MeS}_A & \\
& \mid (a \in \texttt{MeS}_A \gg\!= (k \in A \to \texttt{MeS}_B)) \in \texttt{MeS}_B \quad \mid \texttt{ret}\ (a \in A) \in \texttt{MeS}_A & \\
\texttt{MeR} \in \texttt{Type} \to \texttt{Type} = & \mid \texttt{loop}\ (k \in X \to \texttt{MeR}_X)\ (x \in X) \in \texttt{MeR}_{\texttt{unit}} & \\
& \mid (l \in \texttt{MeR}_A) \oplus (r \in \texttt{MeR}_A) \in \texttt{MeR}_A \quad \mid \texttt{call}\ (p \in \texttt{MeS}_A) \in \texttt{MeR}_A & \\
& \mid (a \in \texttt{MeR}_A \gg\!= (k \in A \to \texttt{MeR}_B)) \in \texttt{MeR}_B \quad \mid \texttt{ret}\ (a \in A) \in \texttt{MeR}_A &
\end{array}
$$

Fig. 25. Process language MeS has read-write access to a security labelled heap. Scheduler language MeR has infinite loops (loop), process calls (call) and nondeterministic choice ($\oplus$).

($l_m \leq l_i$). We prove this *always* property for two interleaved processes at different security levels, *alice* and *bob*. The nondeterministic interleaving of *alice* and *bob* is a superset of all concurrent traces—by proving safety in the interleaving we guarantee safety in all concurrent executions.

The instrumentation handler (h_S), the denotational semantics of the process language (MeS) and scheduler language (MeR), the structural lemmas for MeR, and the complete safety proof can be found in Appendix B. The definition of the languages (Figure 25), ticl structural lemmas (Figure 30), and the safety proof (Figure 31) required 174 lines of Rocq definitions and 242 lines of proofs.

7.3 Distributed Consensus

For the last example, we present a different approach to modeling systems, and a new liveness composition lemma. Instead of defining a programming language syntax, following the steps in Sections 6, 7.1, and 7.2, we model a distributed, message-passing system in the metalanguage of the Rocq proof assistant using ICTrees directly. This shallow-embedding provides a shortcut to the interesting part of the protocol proof, reducing the syntax and denotation overhead.

The goal of the protocol is *leader election*; processes must reach consensus on which process will be the *leader*. Leader election is a common component of many distributed protocols like Paxos [28]. We are interested in the liveness property "eventually a leader is elected". For simplicity we assume there are no network, process, or Byzantine failures. Modeling failures by using ICTree's nondeterminism is entirely possible, but doing so is beyond the scope of this paper.

```
proc (pid ∈ PID_n) ∈ ictree_{h_net, unit} :=
  m ← recv pid;
  match m with
  | C candidate ⇒
    match compare candidate pid with
      | Gt ⇒ send pid (C candidate)
      | Lt ⇒ Ret tt
      | Eq ⇒ send pid (E pid)
    end
  | E leader ⇒ send pid (E leader)
  end.
```

Fig. 26. Leader election process.

Processes (Figure 26) perform message-passing events (send and recv) defined in Figure 32. The messages are delivered in a unidirectional ring (*uniring*) configuration in a clockwise manner, as shown in Figure 27. Process scheduling is also in a uniring, following the same pattern. Each process sends and receives one of two kinds of messages: proposing a candidate PID (C_i) and announcing a leader (E_i). The formal definition of messages and mailboxes are in Appendix C. There are two distinct phases in this leader election protocol:

(1) *Aggregating candidate nominations*: initially, every process (*pid*) self-nominates to be the leader. Processes receive a candidate message ($C_{candidate}$). If the candidate PID received

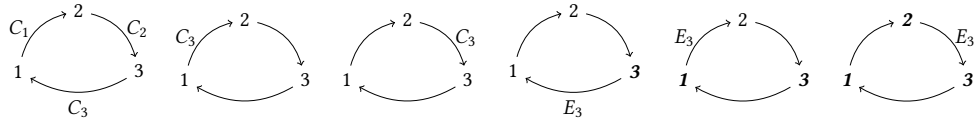

Fig. 27. A unidirectional ring of three processes running the leader election protocol starting at PID= 1.

is greater than the process' own PID ($pid < candidate$), the message is propagated. If the candidate PID is less, the message is dropped.

(2) *Announcing the leader*: if a process (pid) receives their own candidacy message back (C_{pid}), they announce themselves the elected leader (E_{pid}). A process that receives an election announcement (E_{leader}) propagates it. The protocol diverges in the end, sending the leader announcement in cycles forever. The process with the highest PID will always be the leader.

So far in `ticl` we demonstrated uses of modular liveness rules following the structure of programs (e.g., loops). In the protocol in Figure 27 the "modules"—the logical parts of the problem we identified as basic building blocks—are the phases of the protocol, not the scheduler loop. Using the *liveness* lemma (IterAU$_L$) would require establishing an invariant and variant that apply to both phases of the protocol, which is quite challenging and not very modular.

The following theorem illustrates a new liveness composition rule we call *liveness split*:

Theorem 5 (Liveness split).

$$\frac{\begin{array}{l}\mathcal{R}\ i\ w \qquad (\forall\ i\ w,\ \mathcal{R}_I\ i\ w \to \langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle)\\ \forall\ i, w,\ \mathcal{R}\ i\ w \to\\ \quad \langle\ \texttt{k i},\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ (\lambda\ lr\ w'\ \Rightarrow\ \exists\ i',\ lr = \texttt{inl}\ i'\ \wedge\ \mathcal{R}_I\ i'\ w'\ \rangle\ \ \vee\\ \quad \langle\ \texttt{k i},\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ (\lambda\ lr\ w'\ \Rightarrow\ \exists\ i',\ lr = \texttt{inl}\ i'\ \wedge\ \mathcal{R}\ i'\ w'\ \wedge\ f\ i'\ w' < f\ i\ w)\ \rangle\end{array}}{\langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle}\ \textit{SplitAU}_{L,\mathbb{N}}$$

Liveness split breaks up a liveness proof to two parts, before and after a user-defined intermediate point. This reduces a proof of loop liveness to two "smaller" liveness proofs, connected together by an intermediate relation ($\mathcal{R}_I$). Those smaller liveness proofs have smaller, simpler ranking functions (f). What is left afterwards is the same liveness proof we started with, but starting from a better position ($\mathcal{R}_I\ i\ w \to \langle\ \texttt{iter k i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle$). One can continue splitting liveness proofs in this way, by specifying convenient intermediate relations $\mathcal{R}_I$ and ranking functions f. The notion of modularity for liveness proofs extends beyond program structures (i.e: loops), to logical structures, like the phases of the leader election protocol.

The complete liveness proof for the leader election protocol is given in Appendix C. Verification of the protocol required 123 lines of Rocq definitions and 115 lines of proofs.

8 Discussion and related work

`Ticl` is a temporal logic for mechanized, modular verification of safety and liveness properties over effectful, nondeterministic, and potentially nonterminating programs. Its salient aspect is that it can prove general temporal properties, over any programming language denoting to coinductive trees [10, 25, 29, 39, 41, 50, 52, 56], with high-level lemmas and without the bureaucracy of (co-)inductive proofs.

8.1 Comparison with ITrees and CTrees

`ICTrees` are a computational model that is more expressive than Interaction Trees (ITrees) [50] but less expressive than Choice Trees (CTrees) [10]. We introduce this intermediate model of computation, instead of adopting CTrees, because CTrees support two types of nondeterminism but we were only able to prove key lemmas of `ticl` formulas for one of them. In particular, CTrees support *stepping choices* and *delayed choices*. In the context of labelled transition systems (LTS)—which give CTrees their operational semantics—stepping choices correspond to τ *transitions* (they model internal actions that do not change the observable behavior of the system), while delayed choices do not correspond to a transition and defer to a transition in a child node. Delayed choices are important for modeling the equational theory of certain models of concurrency, such as CCS [35]. `ICTrees` support stepping choices but not delayed choices[6].

8.2 Comparison with LTL, CTL, and TLA

The main difference between `ticl` and temporal logics like LTL [36], CTL [19], and TLA [27] is that `ticl` is designed with composition as a guiding principle. As a result, `ticl` needs to handle both finite programs with postconditions, and infinite programs. In more detail, `Ticl`'s treatment of termination is fundamentally different from LTL, CTL and TLA, which assume a total Kripke transition relation (i.e: $\forall\ m,\ \exists\ m', R\ m\ m'$) [15, 20]. `Ticl` uses a non-total transition relation for `ICTree` structures (Section 4.1), a variation on finite trace LTL [5, 14]. However, finite trace logics do not support infinite traces and the *always* operator, and their support for postcondition specifications is limited—`ticl` uses the proof assistants metalanguage to describe complex postconditions.

`Ticl` supports both finite and infinite properties, all CTL operators (φ in Section 4.2) and complex postconditions (ψ_X). By building on recent advances in monadic, coinductive structures [8, 10, 50], `ticl` proof composition follows the monadic composition lemmas. Specifically, it allows `Ret` v to transition to a nullary state, ensuring proper sequencing with continuation k in one step, as required by the monad laws (Figure5). Although there are well-known embeddings of CTL and TLA in the Rocq proof assistant [9, 16], our different approach to modularity and the large number of structural proof lemmas we discovered as a result (Figure 14) indicates this is a still unexplored area of research.

8.3 Comparison with program logics

Comparing `ticl` to existing program logics is straightforward, as those are usually transparent on which property classes they target. This is reflected in their choice of inductive or coinductive big-step semantics, which ties the logic to *eventally* or *always* properties, with no possibility to prove the other class in the future. While current approaches excel within those boundaries, none offer a general, compositional solution for proving arbitrary temporal properties like *always-eventually*.

Iris and Transfinite Iris: Iris [24] is a concurrent-separation logic framework for Rocq that uses step-indexed logical relations to prove safety properties of concurrent programs. The recent extension *Transfinite Iris* [44] extends the step-indexing relation from the naturals to ordinals, allowing total-correctness properties to be proved by transfinite induction. A fundamental limitation of step-indexing is that there is only one index; in the case of "always-eventually" properties, a hierarchy of induction and transfinite induction proofs are required—this hierarchy is implicit in the definition of $\vDash_{L,R}$ in `ticl` (Figure 12). At the same time, `ticl`, unlike Iris, has no facilities for

[6]It remains an open question whether `ticl` could support CTrees. The presence of delayed choice nodes (`BrD`) in CTrees makes structural proofs of $t \gg\!= k$ (bind), where t can make transitive, nondeterministic choices, much harder—specifically, it is not clear what is the inductive invariant on t that characterizes the result of the entire bind.

separation logic. One can imagine having the "best of both worlds", combining the separation logic reasoning of Iris and temporal reasoning of `ticl`.

Fair operational semantics: Lee et al. [29] recognize the limited support for liveness properties in mechanized formal verification and propose an operational semantics for fairness (FOS). FOS uses implicit counters for *bad* events and defines operational semantics that prove no infinite chain of *bad* events happens. FOS provides comprehensive support for the specific case of *binary* fairness (*good* vs *bad* events), but limited support for general temporal specifications, like safety, liveness and termination. As with Iris, it would be interesting to combine that approach with `ticl`.

Maude: The Maude language and Temporal Rewriting Logic (TLR) [33, 34] recognize the benefits of structural approaches (namely term rewriting) to temporal logic verification. In `ticl` we enable term rewriting with *up-to-tau equivalence* under a temporal context (Section 3.1). However, Maude operates on the level of models, not on the level of executable programs. This creates a verification gap between the executable code and target properties. Finally proof composition in Maude is not modular in the sense of Hoare Logic and `ticl`.

Dijkstra monads: Several works on Dijkstra monads target partial-correctness properties in the style of weakest preconditions [1, 32, 42, 47]. Recent work targets total-correctness properties like "always" [42] but not general temporal properties like liveness.

Synthesising ranking functions: Yao et al. [51] propose an automated synthesis procedure for ranking functions, specialized to proving liveness properties in a class of distributed systems. Similar to model checking, the systems are described as specifications not as implementations which is different from `ticl`. At the same time, automated synthesis of ranking functions is a particularly attractive feature for `ticl`, as they can be used with `ticl` lemmas like $\textsc{IterAU}_{L,\mathbb{N}}$ (Figure 16) to get mostly automated, formal proofs of liveness.

8.4 General Liveness Properties and Completeness

Our focus with `ticl` is on providing a convenient and useful temporal logic. However, its compositionality does not come without a cost: just as with all the other (standard) temporal logics described above, `ticl`'s temporal operators are not *complete*. Classic results from automata theory [48] show that there exist liveness properties that cannot be expressed solely via temporal-logic combinators of the kind supported by `ticl`. To achieve completeness, one would instead have to use alternative means of specifying the desired liveness properties, such as with Büchi automata [2]. But reasoning about the liveness properties expressible by such Büchi automata can, in the limit, require using arbitrarily complex well-foundedness arguments. We can therefore think of temporal logics, generally, as hiding that complexity for the common case where the property of interest is expressible in the logic. In the case of `ticl`, there are a few more subtleties, however: first, its notion of definable observations, which define the set of predicates for the logic, can be an *infinite* set (in contrast to much prior work that uses a finite set of observations), and second, because it is embedded in the Rocq framework, one could always fall back on raw coinductive proofs about some liveness property. It seems possible to extend `ticl` with an "escape hatch" mechanism that would let such proofs act as "axioms" from the point of view of `ticl`'s logic. This would, in theory, recover completeness at the expense of more manual user effort.

8.5 Conclusion

In this work we ask: is it possible to write modular proofs about programs in a general temporal logic akin to proofs in Hoare logic? We believe we have answered affirmatively, and in the process developed Temporal Interaction and Choice Logic (`ticl`), a specification language capable of expressing general liveness and safety properties (we summarize `ticl` in Figure 10). Along the way, we also designed an extensive metatheory of structural lemmas (Figures 14, 15, 16, 19) that

encapsulate complex (co-)inductive proofs to simple rule application. We applied `ticl` to several examples from the T2 CTL benchmark suite [6] and in three examples inspired from computer systems as a way to demonstrate the metatheory in action.

Acknowledgments

This work was funded in part by NSF Grants CCF-2326576, CCF-2124184, CNS-2107147, CNS-2321726.

References

[1] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[2] Bowen Alpern and Fred B Schneider. 1987. Recognizing safety and liveness. *Distributed computing* 2, 3 (1987), 117–126.

[3] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. 2002. Alternating-time temporal logic. *Journal of the ACM* 49, 5 (2002).

[4] Andrew W Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001).

[5] Alessandro Artale, Andrea Mazzullo, and Ana Ozaki. 2019. Do You Need Infinite Time?. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

[6] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: temporal property verification. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[7] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical computer science* 59, 1-2 (1988).

[8] Venanzio Capretta. 2005. General recursion via coinductive types. *Logical Methods in Computer Science* 1 (2005).

[9] Tej Chajed. 2024. *coq-tla: Embedding the TLA Logic in Coq*. https://github.com/tchajed/coq-tla

[10] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 7, POPL (2023).

[11] Nicolas Chappe, Ludovic Henrio, and Yannick Zakowski. 2025. Monadic Interpreters for Concurrent Memory Models: Executable Semantics of a Concurrent Subset of LLVM IR. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 283–298.

[12] Adam Chlipala. 2021. Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl). *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–28.

[13] Alessandro Cimatti, Alberto Griggio, and Gianluca Redondi. 2022. Verification of SMT systems with quantifiers. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 154–170.

[14] Giuseppe De Giacomo, Moshe Y Vardi, et al. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

[15] Rocco De Nicola and Frits Vaandrager. 1990. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*.

[16] Christian Doczkal and Gert Smolka. 2016. Completeness and decidability results for CTL in constructive type theory. *Journal of Automated Reasoning* 56 (2016).

[17] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2021).

[18] E. Allen EMERSON. 1990. CHAPTER 16 - Temporal and Modal Logic. In *Formal Models and Semantics*, JAN VAN LEEUWEN (Ed.). Elsevier, Amsterdam, 995–1072. https://doi.org/10.1016/B978-0-444-88074-1.50021-4

[19] E Allen Emerson and Edmund M Clarke. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming* 2, 3 (1982).

[20] E Allen Emerson and Joseph Y Halpern. 1986. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)* 33, 1 (1986), 151–178.

[21] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving liveness of parameterized programs. In *Proceedings of the Annual ACM/IEEE Symposium on Logic in Computer Science*.

[22] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the Symposium on Operating*

Systems Principles (SOSP).

[23] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The power of parameterization in coinductive proof. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

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

[25] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*.

[26] Dexter Kozen and Rohit Parikh. 1984. A decision procedure for the propositional μ-calculus. In *Logics of Programs: Workshop, Carnegie Mellon University Pittsburgh, PA, June 6–8, 1983*. Springer.

[27] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994).

[28] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001).

[29] Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. 2023. Fair operational semantics. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023).

[30] Mohsen Lesani, Li-Yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: Verified Transactional Objects. *Proceedings of the ACM on Programming Languages* OOPSLA (2022).

[31] Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[32] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019).

[33] José Meseguer. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science* 96, 1 (1992), 73–155.

[34] José Meseguer. 2008. The temporal logic of rewriting: A gentle introduction. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*. Springer, 354–382.

[35] Robin Milner. 1980. *A calculus of communicating systems*. Springer.

[36] Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*.

[37] Amir Pnueli and Elad Shahar. 2000. Liveness and acceleration in parameterized verification. In *International Conference on Computer Aided Verification (CAV)*. Springer.

[38] Damien Pous. 2016. Coinduction all the way up. In *Proceedings of the Annual ACM/IEEE Symposium on Logic in Computer Science*.

[39] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. 2023. Semantics for Noninterference with Interaction Trees. In *Proceedings of the 37th Annual European Conference on Object-Oriented Programming (ECOOP 2023)*.

[40] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. 2023. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

[41] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. 2023. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Vol. 263.

[42] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

[43] A Prasad Sistla, Moshe Y Vardi, and Pierre Wolper. 1987. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science* 49, 2-3 (1987).

[44] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[45] Gadi Tellez and James Brotherston. 2017. Automatically verifying temporal properties of pointer programs with cyclic proof. In *In proceedings of the International Conference on Automated Deduction (CADE)*.

[46] The Rocq Development Team. 2025. The Rocq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/master/refman/.

[47] Théo Winterhalter, Cezar-Constantin Andrici, C Hriţcu, Kenji Maillard, G Martínez, and Exequiel Rivas. 2022. Partial dijkstra monads for all. In *Proceedings of the International Conference on Types for Proofs and Programs (TYPES)*.

[48] Pierre Wolper. 1983. Temporal logic can be more expressive. *Information and Control* 56, 1 (1983), 72–99. https://doi.org/10.1016/S0019-9958(83)80051-5
[49] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371119
[50] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019).
[51] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2024. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. *Proceedings of the ACM on Programming Languages* 8, POPL (2024).
[52] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal Reasoning About Layered Monadic Interpreters. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022).
[53] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced research working conference on correct hardware design and verification methods*.
[54] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and S. Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proceedings of the ACM on Programming Languages* 5 (08 2021).
[55] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*.
[56] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*.

A Coinductive Proofs and Up-to Principles in Rocq

In this appendix we focus on the low-level coinduction constructs in `ticl` used to define the *forever* operators `AG`, `EG`. The implementation details of coinductive structures and proofs in proof assistants differs. We focus on the Rocq proof assistant, where the infrastructure for coinductive proofs in provided by external libraries [23, 38, 55]. `Ticl` relies on the `coinduction` library by Damien Pous [38] to define greatest fixpoints over the complete lattice of Rocq propositions.

The primary construction offered by the library is a greatest fixpoint operator ($\texttt{gfp}\ b : X$) for any complete lattice X and monotone endofunction $b : X \to X$. Specifically, the library proves Rocq propositions form a complete lattice, as do any functions from an arbitrary type into a complete lattice. Consequently, coinductive relations of arbitrary arity over arbitrary types can be constructed using this combinator. In `ticl`, we target coinductive predicates over `ICTrees` and worlds so we work in the complete lattice $\texttt{ictree}_{E,\,X} \to \mathcal{W}_E \to \mathbb{P}$.

The `coinduction` library [38] provides tactic support for coinductive proofs based on Knaster-Tarski's theorem: any post-fixpoint is below the greatest fixpoint. Given an endofunction b, a (sound) enhanced coinduction principle, also known as an up-to principle, involves an additional function $f : X \to X$ allowing one to work with $b \circ f$ (the composition of b with f) instead of b: any post-fixpoint of $b \circ f$ is below the greatest fixpoint of f. Practically, this gives the user access to a new proof principle. Rather than needing to "fall back" precisely into their coinduction hypothesis after "stepping" through b, they may first apply f.

In Figure 28 we give the up-to-principles for coinduction proofs in `ticl`. The $\textsc{upto}^{UP}(\mathit{equiv})$ principle is used to show $\textsc{upto}^{UP}(\mathit{equiv}) \le \lambda\, t.\, \texttt{gfp}\ (\texttt{anc}\ \varphi)\ t$, meaning equivalent trees (abstracting over the exact equivalence relation) satisfy the same $\texttt{AG}\ \varphi$ formula (similarly $\texttt{EG}\ \varphi$). Note, $\textsc{upto}^{UP}(\mathit{equiv})$ is sufficiently general; any equivalence relation `equiv` satisfying the ExEquiv lemma in Figure 9 can be used in place of up-to-tau equivalence ($\sim$).

Up-to-principles $\textsc{bindAG}^{UP}(\varphi,\ P_X)$, $\textsc{bindEG}^{UP}(\varphi,\ P_X)$, parametrized by a prefix formula φ and a postcondition $P_X \in X \to \mathcal{W}_E \to \mathbb{P}$ are used to prove the bind lemmas in Figure 15. Specifically, by showing the bind principle is under the greatest fixpoint $\textsc{bindAG}^{UP}(\varphi,\ P_X) \le \texttt{gfp}\ (\texttt{anc}\ \varphi)$ we reduce a coinductive proof $\langle\ x \leftarrow t;;\ k\ x,\ w \vDash_L \texttt{AG}\ \varphi\ \rangle$ to an *inductive* proof on the finite prefix

$\langle\ t,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ P_X\ \rangle$ and a coinductive proof about its continuation k. The iteration lemmas in Figure 16 reduce to using the same bind up-to-principles.

B Secure concurrent shared memory proof

This appendix provides the complete formalization of the security-typed memory system introduced in Section 7.2. We present the instrumentation handler ($h_{\mathcal{S}}$), denotational semantics ($[\![\_]\!]_{\mathcal{S}}$, $[\![\_]\!]_{\mathbb{R}}$), and proof lemmas (Figure 30) for proving confidentiality properties in concurrent memory access scenarios.

The goal for this example is to show confindentiality; if every read (`read` $l_i\ x$) has an instruction label l_i, and accesses a memory label with l_m, then it should always be true that $l_m \leq l_i$.

The two processes are in the bottom of Figure 31, `alice` has high-security access and *writes* to *odd* numbered memory indices, while bob who has low-security access and *reads* from *even* memory indices. The two processes are written in the `MeS` language—with reads, writes and conditionals—while the interleaving scheduler is written in the scheduler language `MeS` (Figure 25)—with infinite loops, nondeterministic choice, sequential composition and calls to `MeS`. Having a different scheduler language from the process language simplifies the example, however, both languages denote to a common coinductive structure ($\mathtt{ictree}_{E_{\mathcal{S}}}$) in Figure 29.

As bob can potentially read every *even* indexed location, we must ensure the starting state has no high-security, even locations to begin with; this is the `noleak` invariant on states and indices (Figure 31). The proof in Figure. 31 starts by using the *invariance* lemma $\textsc{Loop}_R\mathsf{AG}$ (Figure 30) with loop invariant $\mathcal{R}$, then two proof obligations remain:

(1) The *loop* satisfies the safety property initially (`obs` $(l_m \leq l_i)$).
(2) The *loop body* steps (outer `AX`) then satisfies `obs` $(l_m \leq l_i)$ until termination, at which point loop invariant $\mathcal{R}$ is satisfied at the end state.

The invariance rule ($\textsc{Loop}_R\mathsf{AG}$) abstracts the complex coinductive proof, proved once and for all by lemma $\textsc{IterAG}$, adapted to language `MeS` in a few lines of Rocq syntactic manipulations. The rest of the proof in Figure 31 is straightforward. Proceed by examining both cases of the nondeterministic choice ((`alice` $x\ i$) $\oplus$ (`bob` i)) using the choice rule $\textsc{Br}_R\mathsf{AX}$. We consider both cases due to the universal quantifier(`AX`). The two subproofs proceed by case analysis on the evenness of memory index i and using the low-level theory of finite maps.

C Distributed Consensus proof

This appendix provides the complete formalization for the leader election distributed consensus proof in Section 7.3. A distributed, message-passing system is encoded using `ICTrees` directly and the liveness property "eventually a consensus is reached" is proved using the `ticl` structural lemmas (Section 16).

$$
\begin{aligned}
\textsc{upto}^{UP}(\mathit{equiv})\ \mathcal{R} &\triangleq \{t \mid \exists\, t',\ \mathsf{equiv}\ t\ t' \wedge \mathcal{R}\ t'\} \\
\textsc{bindAG}^{UP}(\varphi, P_X)\ \mathcal{R} &\triangleq \{(t \gg\!= k, w) \mid \langle\ t,\ w \vDash_R\ \varphi\ \mathsf{AU}\ \mathsf{AX}\ \mathsf{done}\ P_X\ \rangle \\
&\qquad\qquad \wedge\ (\forall\, x, w,\ P_X\ x\ w \to \mathcal{R}\ (k\ x)\ w)\} \\
\textsc{bindEG}^{UP}(\varphi, P_X)\ \mathcal{R} &\triangleq \{(t \gg\!= k, w) \mid \langle\ t,\ w \vDash_R\ \varphi\ \mathsf{EU}\ \mathsf{EX}\ \mathsf{done}\ P_X\ \rangle \\
&\qquad\qquad \wedge\ (\forall\, x, w,\ P_X\ x\ w \to \mathcal{R}\ (k\ x)\ w)\}
\end{aligned}
$$

Fig. 28. Up-to-principles for coinductive AG, EG proofs.

$$\mathsf{h}_{\mathcal{S}} \in E_{\mathcal{S}} \rightsquigarrow \mathtt{InstrM}_{\mathcal{M}_{\mathcal{S}},(\mathcal{S}*\mathcal{S})}$$

$$h_{\mathcal{S}}\ (\mathtt{Read}\ l_i\ x \in E_S)\ (m \in \mathcal{M}_{\mathcal{S}}) \ =\ \begin{cases} \mathtt{log}\ (l_m, l_i)\ ;;\ \mathtt{Ret}\ (v, m), & \text{if } m[x] = \mathtt{Some}(l_m, v) \\ \mathtt{Ret}\ (0, m), & \text{otherwise} \end{cases}$$

$$h_{\mathcal{S}}\ (\mathtt{Write}\ l_i\ x\ v \in E_S)\ (m \in \mathcal{M}_{\mathcal{S}}) \ =\ \mathtt{Ret}\ ((), (x \hookrightarrow v) \cup m)$$

$$[\![\_]\!]_{\mathcal{S}} \in \mathtt{MeS}_A \to \mathtt{ictree}_{E_{\mathcal{S}},\,A}$$

$$[\![\mathtt{read}\ l_i\ n]\!]_{\mathcal{S}} \ =\ \mathtt{trigger}\ (\mathtt{Read}\ l_i\ n), \quad [\![\mathtt{write}\ l_i\ n\ v]\!]_{\mathcal{S}} \ =\ \mathtt{trigger}\ (\mathtt{Write}\ l_i\ n\ v),$$

$$[\![\textbf{if}\ c\ \textbf{then}\ t\ \textbf{else}\ u]\!]_{\mathcal{S}} \ =\ \begin{cases} [\![t]\!]_{\mathcal{S}}, & \text{if } c \\ [\![u]\!]_{\mathcal{S}}, & \text{otherwise} \end{cases}$$

$$[\![x \gg\!= k]\!]_{\mathcal{S}} \ =\ [\![x]\!]_{\mathcal{S}} \gg\!= (\lambda\ a \Rightarrow [\![k\ a]\!]_{\mathcal{S}}), \quad [\![\mathtt{ret}\ x]\!]_{\mathcal{S}} \ =\ \mathtt{Ret}\ x$$

$$[\![\_]\!]_{\mathtt{R}} \in \mathtt{MeR}_A \to \mathtt{ictree}_{E_{\mathcal{S}},\,A}$$

$$[\![\mathtt{loop}\ k\ x]\!]_{\mathtt{R}} \ =\ \mathtt{iter}\ (\lambda\ \mathtt{x}' \Rightarrow [\![\mathtt{k}\ \mathtt{x}']\!]_{\mathcal{S}} \gg\!= (\lambda\ \mathtt{v} \Rightarrow \mathtt{Ret}\ (\mathtt{inl}\ \mathtt{v})))\ \mathtt{x}, \quad [\![l \oplus r]\!]_{\mathtt{R}} \ =\ [\![l]\!]_{\mathtt{R}} \oplus [\![r]\!]_{\mathtt{R}},$$

$$[\![\mathtt{call}\ p]\!]_{\mathtt{R}} \ =\ [\![p]\!]_{\mathcal{S}}, \quad [\![\mathtt{ret}\ x]\!]_{\mathtt{R}} \ =\ \mathtt{Ret}\ x, \quad [\![x \gg\!= k]\!]_{\mathtt{R}} \ =\ [\![x]\!]_{\mathtt{R}} \gg\!= (\lambda\ a \Rightarrow [\![k\ a]\!]_{\mathtt{R}})$$

$$[(t \in \mathtt{MeS}_A),\ (m \in \mathcal{M}_{\mathcal{S}}),\ (l \in \mathcal{S} * \mathcal{S}) \Vdash_{LR}\ p]_{\mathcal{S}} \in \mathbb{P} \ =\ \langle\ \mathtt{instr}\ h_{\mathcal{S}}\ [\![p]\!]_{\mathcal{S}}\ m,\ \mathtt{Obs}\ (\mathtt{Log}\ l)\ ()\ \vDash_{LR}\ p\ \rangle$$

$$[(r \in \mathtt{MeR}_A),\ (m \in \mathcal{M}_{\mathcal{S}}),\ (l \in \mathcal{S} * \mathcal{S}) \Vdash_{LR}\ p]_{R} \in \mathbb{P} \ =\ \langle\ \mathtt{instr}\ h_{\mathcal{S}}\ [\![r]\!]_{\mathtt{R}}\ m,\ \mathtt{Obs}\ (\mathtt{Log}\ l)\ ()\ \vDash_{LR}\ p\ \rangle$$

Fig. 29. Denotation of process and scheduler languages MeS and MeR, with a tagged heap. Observes memory labels and instruction labels to prove safety.

$$\frac{[a,\ m,\ l \Vdash_{LR}\ p]_R \quad [b,\ m,\ l \Vdash_{LR}\ p]_R}{[a \oplus b,\ m,\ l \Vdash_{LR}\ \mathtt{AX}\ p]_R}\ \textsc{Br}_R\textsc{AX} \qquad \frac{\mathcal{R}\ i\ m\ l \to \forall\ i, m, l,\ \mathcal{R}\ m\ l \to [\mathtt{loop}\ k\ i,\ m,\ l \Vdash_L\ \varphi]_R \wedge [k\ i,\ m,\ l \Vdash_R\ \mathtt{AX}\ (\varphi\ \mathtt{AU}\ \mathtt{AX}\ \mathtt{done}\ \mathcal{R})]_R}{[\mathtt{loop}\ k\ i,\ m,\ l \Vdash_L\ \mathtt{AG}\ \varphi]_R}\ \textsc{Loop}_R\textsc{AG}$$

Fig. 30. Representative `ticl` lemmas for process and scheduler languages MeS and MeR.

$$\frac{\begin{array}{l} \forall\ i, m, l_m, l_i,\ \mathcal{R}\ i\ m\ (l_m, l_i) \to \\ [\textbf{\textit{alice}}\ x\ i,\ m,\ (l_m, l_i) \Vdash_R\ \mathtt{obs}\ (l_m \le l_i)\ \mathtt{AU}\ \mathtt{AX}\ \mathtt{done}\ \mathcal{R}]_R \checkmark \end{array} \quad \begin{array}{l} \forall\ i, m, l_m, l_i,\ \mathcal{R}\ i\ m\ (l_m, l_i) \to \\ [\textbf{\textit{bob}}\ i,\ m,\ (l_m, l_i) \Vdash_R\ \mathtt{obs}\ (l_m \le l_i)\ \mathtt{AU}\ \mathtt{AX}\ \mathtt{done}\ \mathcal{R}]_R \checkmark \end{array}}{\begin{array}{l} \forall\ i, m, l_m, l_i,\ \mathcal{R}\ i\ m\ (l_m, l_i) \to \\ \mathcal{R}\ \_\ m\ (l_m, l_i)\ :=\ l_m \le l_i\ \wedge\ \forall\ i, \textbf{\textit{noleak}}\ i\ m \\ \mathcal{R}\ 0\ m\ (l_m, l_i) \checkmark \end{array} \quad \begin{array}{l} \left[\begin{array}{l} \textbf{loop}\ (\lambda\ i \Rightarrow \\ \quad (\textbf{\textit{alice}}\ x\ i) \oplus (\textbf{\textit{bob}}\ i); \\ \quad \mathtt{ret}\ (i+1); \\ )\ i \end{array}\right., \ m,\ (l_m, l_i) \Vdash_L \mathtt{obs}\ (l_m \le l_i) \Bigg]_R \checkmark \\ \wedge\ [(\textbf{\textit{alice}}\ x\ i) \oplus (\textbf{\textit{bob}}\ i);\ \mathtt{ret}\ (i+1),\ m,\ (l_m, l_i) \Vdash_R\ \mathtt{AX}\ (\mathtt{obs}\ (l_m \le l_i)\ \mathtt{AU}\ \mathtt{AX}\ \mathtt{done}\ \mathcal{R})]_R \end{array}}\ \textsc{Loop}_R\textsc{AG}$$

$$\begin{array}{l} \textbf{\textit{alice}}\ (x\ i \in \mathbb{N}) := \\ \quad \textbf{if}\ (\mathtt{even}\ i)\ \textbf{then} \\ \quad\quad \mathtt{write}\ H\ (i+1)\ x \\ \quad \textbf{else} \\ \quad\quad \mathtt{write}\ H\ i\ x \end{array} \qquad \begin{array}{l} \textbf{\textit{bob}}\ (i \in \mathbb{N}) := \\ \quad \textbf{if}\ (\mathtt{even}\ i)\ \textbf{then} \\ \quad\quad \mathtt{read}\ L\ i \\ \quad \textbf{else} \\ \quad\quad \mathtt{read}\ L\ (i+1) \end{array}$$

$$\begin{array}{l} \textbf{\textit{noleak}}\ (i \in \mathbb{N})\ (m \in \mathcal{M}_{\mathcal{S}}) := \\ \quad \mathtt{even}\ i \to \exists\ v, m[i] = \mathtt{Some}((L, v)) \end{array} \qquad \begin{array}{l} \forall\ x, m, l_m, l_i,\ l_m \le l_i \to \forall\ i,\ \textbf{\textit{noleak}}\ i\ m \to \\ \left[\begin{array}{l} \textbf{loop}\ (\lambda\ i \Rightarrow \\ \quad (\textbf{\textit{alice}}\ x\ i) \oplus (\textbf{\textit{bob}}\ i); \\ \quad \mathtt{ret}\ (i+1); \\ )\ 0 \end{array}\right., \ m,\ (l_m, l_i) \Vdash_L \mathtt{AG}\ \mathtt{obs}\ (l_m \le l_i) \Bigg]_R \end{array}$$

Fig. 31. `Ticl` safety proof for the concurrent secure heap. The goal is to show that `read` instructions only access memory locations with a security level lower-or-equal than their own.

Each process in the leader election protocol (Figure 26) is assigned a process identifier ($pid \in \mathtt{PID}_n$). Message-passing events (`send` and `recv`) are defined in Figure 32. Their semantic meaning by an

$E_{\mathsf{net}} \in \mathtt{Type} \to \mathtt{Type} \;=\; \mid \mathsf{Send}(id \in \mathtt{PID}_n)(m \in \mathtt{Msg}_n) \;\mid \mathsf{Recv}(id \in \mathtt{PID}_n)$

$h_{\mathsf{net}} \in E_{\mathsf{net}} \rightsquigarrow \mathtt{InstrM}_{[\mathtt{Msg}_n]_n, \mathtt{Msg}_n}$
$h_{\mathsf{net}}\ (\mathsf{Send}\ id\ msg)\ (m \in [\mathtt{Msg}_n]_n) \;=\; \mathsf{Ret}\ ((), m[id\ +\ 1\ \%\ n] := msg)$
$h_{\mathsf{net}}\ (\mathsf{Recv}\ id)\ (m \in [\mathtt{Msg}_n]_n) \;=\; \mathsf{log}\ (m[id]) \;;;\; \mathsf{Ret}\ (m[id], m)$

Fig. 32. Message-passing events (E_{net}) parametrized by the caller id and their instrumentation (h_{net}).

$\mathtt{PID}_n \in \mathtt{Type} \;=\; \mathtt{fin}\ n$
$\mathtt{Msg}_n \in \mathtt{Type} \;=\; \mid C\ (p \in \mathtt{PID}_n) \;\mid E\ (p \in \mathtt{PID}_n)$
$[\mathtt{Msg}_n]_n \in \mathtt{Type} \;=\; \mathtt{Vector}\ n\ \mathtt{Msg}_n$
$(m \in [\mathtt{Msg}_n]_n)[p \in \mathtt{PID}_n] \in \mathtt{Msg}_n,$
$(m \in [\mathtt{Msg}_n]_n)[p \in \mathtt{PID}_n] := (msg \in \mathtt{Msg}_n) \in [\mathtt{Msg}_n]_n$

Fig. 33. Process identifiers, messages and mailboxes with *get* ($m[p]$) and *set* ($m[p] := msg$) access.

instrumentation handler (h_{net})— messages are delivered in a unidirectional ring (*uniring*) configuration in a clockwise manner, as shown in Figure 27.

Scheduling follows the same pattern. Each process sends and receives one of two kinds of messages: proposing a candidate PID (C_i) and announcing a leader (E_i). The formal definition of messages ($\mathtt{Msg}_n$) and mailboxes ($[\mathtt{Msg}_n]_n$) are in Figure 33. Each process has exactly one message in its mailbox. Messages and process identifiers ($\mathtt{PID}_n$), are both indexed by the number of processes in the system ($n \in \mathbb{N}$).

The liveness proof ("eventually a consensus is reached") in Figure 34 is proven using the *liveness split* lemma (Theorem 5). *Liveness split* divides a liveness proof into two sections, separated by a user-specified intermediate point. This transforms a loop liveness proof into two "smaller" liveness proofs, joined by an intermediate relation ($\mathcal{R}_I$). These reduced liveness proofs utilize more straightforward, compact ranking functions (f). The result is the original liveness proof, but starting from a later position ($\mathcal{R}_I\ i\ w \to \langle\ \mathtt{iter\ k\ i},\ w \vDash_L\ \varphi\ \mathsf{AU}\ \varphi'\ \rangle$). This splitting process can be continued by defining appropriate intermediate relations ($\mathcal{R}_I$) and ranking functions (f). This way, liveness proof modularity extends to logical structures, such as the phases in the leader election protocol (Section 7.3).

The proof in Figure 34 is for three processes ($n = 3$), but the techniques used generalize to any number. Start from the bottom, the goal liveness property is:

Lemma 6 (Eventual leader consensus).

$$\langle\ \mathtt{instr}\ h_{\mathsf{net}} \left(\begin{array}{l} i \leftarrow \mathtt{branch}\ n; \\ \mathtt{iter}\ (\lambda\ i \Rightarrow \\ \quad \mathbf{proc}\ i; \\ \quad \mathtt{Ret\ (inl\ ((}i\ \mathtt{+\ 1)\ \%}n\mathtt{))} \\ )\ i \end{array} \right) [C_3, C_1, C_2],\ \mathtt{Pure} \vDash_L\ \mathtt{AF\ obs}\ E_3\ \rangle$$

The shared state of this system captures the mailboxes starting with $[C_3, C_1, C_2]$, meaning candidacy messages are waiting to be received by their respective process. The ghost-state of the system (see h_{net} in Figure 32) is the last received message ($\mathtt{log}\ m[id]$). Our goal specification (Lemma 6) is to eventually observe $pid = 3$ elected as the leader (E_3). When message E_3 is received at least by one process, it will be infinitely propagated in a clockwise manner until all processes know the leader.

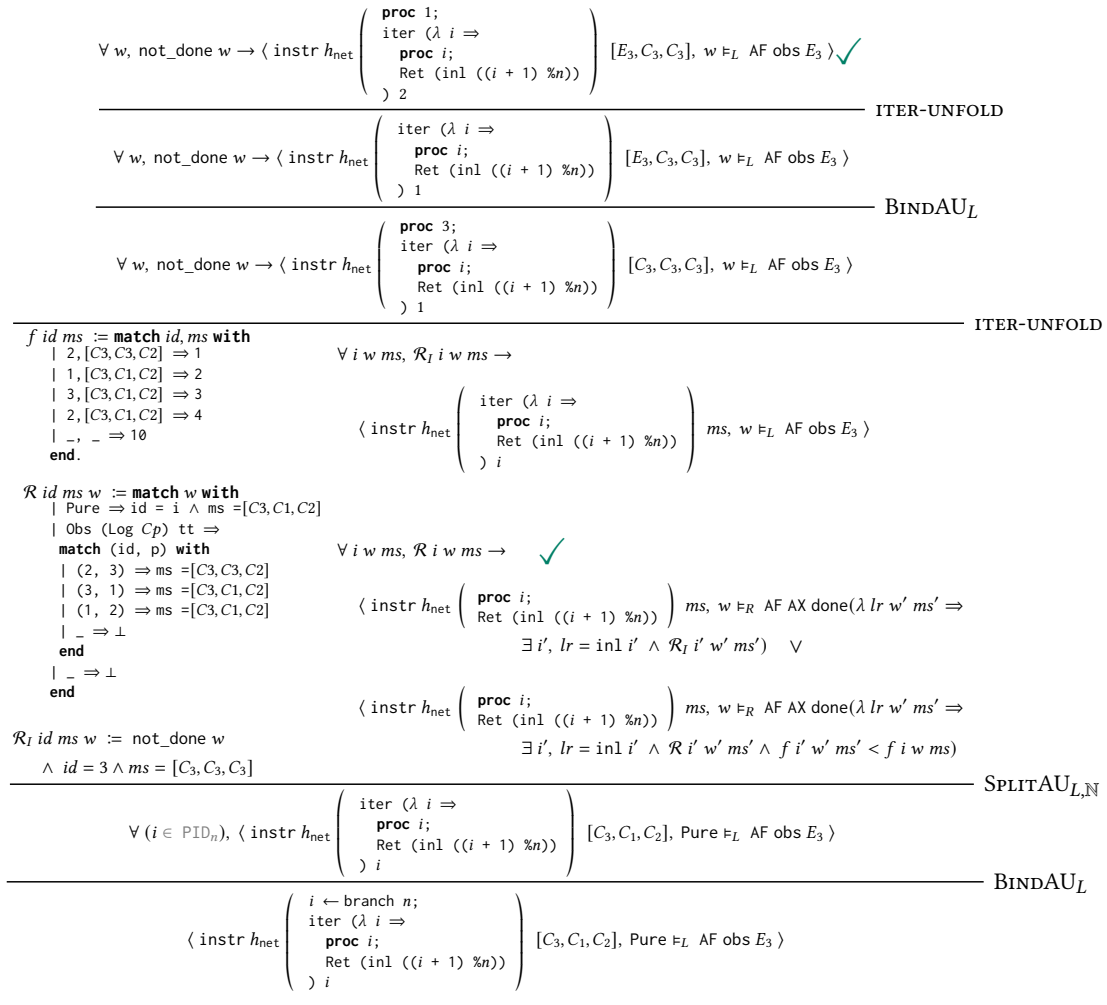

Fig. 34. `Ticl` liveness proof for the distributed consensus example, is split in two subproofs using intermediate relation $\mathcal{R}_I$.

Start by nondeterministically choosing a process to schedule ($i \leftarrow$ `branch` n) first. Applying the sequence lemma (BINDAU$_L$) introduces $i \in \texttt{PID}_3$ to the proof context. We must prove that eventually a leader is elected, regardless of which process the scheduler chooses first. Figure 34 uses liveness split (SPLITAU_$L, \mathbb{N}$) with an intermediate relation ($\mathcal{R}_I$) to mark the end of the candidate aggregation phase. Loop invariant ($\mathcal{R}$) shows that, if we know which process ran last (C_P) and which process is currently running (id) we can guess the state of mailboxes (either $[C_3, C_3, C_2]$ or $[C_3, C_1, C_2]$ or $[C_3, C_1, C_2]$) by checking Figure 27.

By case analysis (on w, id, p) we can show that regardless of the nondeterministic choice of first PID (i), candidate aggregation eventually ends in $\mathcal{R}_I$: The highest PID's candidacy message (C_3) is in every mailbox and it its process is ready to be scheduled ($i = 3$). Establishing this intermediate

goal simplifies the ranking function f and invariant $\mathcal{R}$, which now only need to refer to candidacy messages (C_i), not election announcement messages (E_i).

Now what's left is the remaining liveness proof, starting at a point ($\mathcal{R}_I$) where the mailboxes have fully propagated the candidacy of $pid = 3$ ($[C_3, C_3, C_3]$) and also $i = 3$ is scheduled to run. Using the equational theory of `ICTrees` (Section 4.5) we unfold one iteration of the loop (`proc 3`). Consequently we use the sequencing lemma ($\textsc{BindAU}_L$) to evaluate the proccess with $pid = 3$, updating the mailbox of its neighbor ($pid = 1$). The state after running `proc 3` is $[E_3, C_3, C_3]$ and the next process scheduled to run is $i = 1$. When `proc 1` runs, it will receive the election announcement (E_3), satisfying the liveness property (`obs` E_3) and concluding the proof.

Consequently, from state $\mathcal{R}_I$, `proc 3` runs with C_3 in their mailbox and announces themselves as the new leader (E_3). The next process is then scheduled ($i = 1$), receiving the leader announcement (E_3) and satisfying the goal (`AF obs` E_3).