

EXPRESSIVE SPECIFICATION LANGUAGES FOR MACHINE-CHECKED PROOFS

Eleftherios Ioannidis

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2024

Supervisor of Dissertation

Sebastian Angel

Assistant Professor

Co-Supervisor of Dissertation

Steve Zdancewic

Schlein Family President's Distinguished Professor and Associate Chair

Graduate Group Chairperson

Dissertation Committee

Rajeev Alur

Pratyush Mishra

Stephanie Weirich

Mike Hicks

EXPRESSIVE SPECIFICATION LANGUAGES FOR MACHINE-CHECKED PROOFS

COPYRIGHT

2024

Eleftherios Ioannidis

ABSTRACT

EXPRESSIVE SPECIFICATION LANGUAGES FOR MACHINE-CHECKED PROOFS

Eleftherios Ioannidis

Sebastian Angel

Steve Zdancewic

Formal verification provides mathematical guarantees that computer programs are correct with regards to a specification, preventing errors, outages, and security vulnerabilities. Effective program verification faces two challenges: expressing complex programs and capturing their diverse specifications — from properties expressed with regular expressions, to safety and liveness, to more sophisticated privacy properties such as zero-knowledge. While many expressive formal semantics exist for programming languages, specification languages often lack in expressive power. This dissertation explores how proof systems can be tailored to express rich specifications, proved efficiently with machine-checkable proofs. We begin by introducing SAFA, a novel finite automaton designed to efficiently verify extended regular specifications over large traces and with succinct, checkable proofs. We then present Ticl, a structural temporal logic embedded in the Coq proof assistant. Ticl enables syntax-directed verification of nested safety and liveness properties while maintaining a high level of abstraction and remarkably concise proofs, as demonstrated through verifying programs with job queues, concurrent shared memory, and distributed consensus protocols. Finally, we propose a new research direction with Zippel, a statically-typed programming language and compiler with three goals in mind — easily and safely expressing zero-knowledge proof systems, compiling them to fast, multithreaded implementations and automatically verifying their zero-knowledge properties using techniques information flow security.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF ILLUSTRATIONS	vi
CHAPTER 1 : Introduction	1
1.1 The Promise and Challenge of Program Verification	1
1.2 Expressive formal specifications	2
1.3 Modular machine-checked proofs	2
1.4 Contributions	3
1.5 Dissertation Overview	4
CHAPTER 2 : Background and related work	6
2.1 Modal Logic and Kripke Structures	6
2.2 Finite automata	8
2.3 Extended Regular Expressions	10
2.4 Derivatives of regular expressions	11
CHAPTER 3 : Skipping Alternating Finite Automata (SAFA)	15
3.1 Regular expression specifications	15
3.2 Alternating Finite Automata (AFA)	15
3.3 Supporting Skips	17
3.4 SAFA formal definition	20
3.5 SAFA compiler	21
3.6 SAFA Satisfaction	25
3.7 Conclusion and Future Work	28
CHAPTER 4 : Structural Program Verification with Temporal Logic (Ticl)	32
4.1 Safety and Liveness	32

4.2	Low-level temporal proofs	34
4.3	Definitions	35
4.4	Structural lemmas	46
4.5	Motivating examples	54
4.6	Discussion and Related Work	62
CHAPTER 5 : Proposal: a Language and Compiler for Zero-knowledge Protocols (Zippel) .		66
5.1	Example 1: Hadamard product	67
5.2	Schnorr's ZK protocol	68
5.3	Zippel language	72
5.4	Conclusions and future work	78
5.5	Zippel language grammar	79
CHAPTER 6 : Discussion		82
APPENDIX A : APPENDIX A		85
BIBLIOGRAPHY		88

LIST OF ILLUSTRATIONS

FIGURE 2.1	Deterministic Finite Automata (DFA) have a deterministic transition relation.	9
FIGURE 2.2	Noneterministic Finite Automata (NFA) can chose ϵ transitions.	9
FIGURE 2.3	An AFA alternates between existential and universal states.	10
FIGURE 2.4	Regular expression (regex) syntax. Additional syntactic notations are the wild-card $.$ defined as the character set Σ , n -repetition $r\{n\}$ as $r\{n, n\}$ and the match-all \top as $.*$. We may omit the concatenation operator when it is implied.	10
FIGURE 2.5	The set of strings accepted by a regex r is the <i>language</i> $\mathcal{L}[[r]] \subseteq \Sigma^*$.	11
FIGURE 2.6	The predicate $v(r)$ is true when the regex r accepts the empty string.	12
FIGURE 2.7	Brzowski derivative for a regex r given character $\alpha \in \Sigma$ is $d_\alpha(r)$.	13
FIGURE 3.1	AFA for regex $r = (. * a) \wedge (. * b) \wedge (. \{2, 6\})$.	16
FIGURE 3.2	SAFA for regex $r = (. * a) \wedge (. * b) \wedge (. \{2, 6\})$ over alphabet $\Sigma = \{a, b, c\}$ — edges $*$ mean the skip $\{[0, +\infty)\}$.	21
FIGURE 3.3	Rules for a partial, recursive function $\mathbf{take}_s(r) = (s, r')$ extracting skip s from the head a regex r and returning the tail r' .	21
FIGURE 3.4	The generalized partial regex derivative $\partial_\alpha(r)$ is a positive boolean (DNF) expression over regex.	23
FIGURE 3.5	Interval sets are closed under <i>union</i> ($s \cup s'$), <i>concatenation</i> ($s \cdot s'$), <i>Kleene-star</i> (s^*) and <i>bounded repetition</i> ($s\{n, m\}$).	31
FIGURE 4.1	Program rotate runs forever, pops an element from the head and appends it. The specification (rotate_agaf) is <i>always-eventually</i> x will appear in the head position.	34
FIGURE 4.2	Instrumentation of rotate with initial state $\mathbf{q} ++ [x]$.	35
FIGURE 4.3	Definitions and core combinators for ictree .	36
FIGURE 4.4	Equational theory for ictree with respect to <i>up-to-guard</i> equivalence relation.	37
FIGURE 4.5	Instrumentation of an ictree $_{E, X}$ with \log_W events over state S produces an instrumentation monad $\mathbf{InstrM}_{S, W}$.	38
FIGURE 4.6	Kripke world \mathcal{W}_E parametrized by event type E .	39
FIGURE 4.7	Kripke semantics of ctrees and not_done world predicate.	39
FIGURE 4.8	Derived lemmas for kripke transitions for ictree .	40
FIGURE 4.9	Syntax of ticl prefix formulas (φ), suffix formulas (ψ_x) and useful syntactic notations.	41
FIGURE 4.10	<i>Next</i> (anc , enc) and inductive <i>Until</i> (auc , euc) shallow predicates used to define \models_L, \models_R .	42
FIGURE 4.11	Ticl entailment relations $\models_{L, R}$ by induction on formulas.	43
FIGURE 4.12	Some of the implications and equivalences in ticl . Notation $\Rightarrow_{L, R}, \Leftrightarrow_{L, R}$ and formula metavariables p, q capture both prefix and suffix formulas.	44
FIGURE 4.13	Up-to-principles for coinductive AG , EG proofs.	46
FIGURE 4.14	Library of structural, compositional lemmas for ictree combinators and ticl operators. Symbol \Leftarrow indicates a backward-reasoning lemma and \Leftrightarrow lemmas in both directions.	47

FIGURE 4.15	Representative structural lemmas for nondeterminism and sequential composition of $\mathbf{ictree}_{E, X}$	47
FIGURE 4.16	Representative iteration lemmas for \mathbf{AU} , \mathbf{AG} and $\mathbf{ictree}_{E, X}$	50
FIGURE 4.17	Syntax of a small imperative language \mathbf{StImp} with mutable state and nondeterminism.	50
FIGURE 4.18	Some auxiliary map operations from Coq’s standard library are assumed. . . .	50
FIGURE 4.19	Denotation of \mathbf{StImp} programs to the instrumentation monad $\mathbf{InstrM}_{\mathcal{M}, \mathcal{M}}$ by intermediate interpretation to $\mathbf{ictree}_{\text{state}_{\mathcal{M}}, \text{unit}}$	51
FIGURE 4.20	Representative structural lemmas for language \mathbf{StImp} and \mathbf{ticl} operators \mathbf{AU} , \mathbf{AG} . .	53
FIGURE 4.21	Language \mathbf{StImp}_q extends the language \mathbf{StImp} with a global queue as additional state, operations \mathbf{push} and \mathbf{pop} interact with the queue.	54
FIGURE 4.22	Program \mathbf{drain} runs forever, pops all elements in the queue until it eventually spins on the empty queue. Specification $\mathbf{drain_af}$ is; eventually element x will be observed in the head of the queue.	55
FIGURE 4.23	Proof that \mathbf{drain} eventually observes x in the head position of the queue. The goal is in the bottom, work upwards by applying \mathbf{ticl} structural lemmas and basic Coq tactics. Loop invariant \mathbf{Rinv} is in the upper-left corner and loop variant is queue \mathbf{length}	56
FIGURE 4.24	Beginning of “always-eventually” proof for \mathbf{rotate} . Applying $\mathbf{STITERAG}$ using loop invariant \mathbf{Ri} , leaves two finite proof obligations which are easy to conclude. .	57
FIGURE 4.25	Language \mathbf{StImp}_S extends the language \mathbf{StImp} with a global <i>labelled</i> memory where every address (\mathbb{N}) is tagged with either a <i>high</i> security (H) or <i>low</i> security (L) label.	58
FIGURE 4.26	Alice (<i>High</i> security) writes <i>secret</i> to odd addresses and Bob (<i>Low</i> security) reads from even addresses, take their concurrent interleaving.	59
FIGURE 4.27	Beginning of concurrent shared memory proof. Discharging the scheduler and \mathbf{AG} leaves as proof obligations finite proofs about Alice and Bob.	60
FIGURE 4.28	Process identifiers (\mathbf{PID}_n) and messages (\mathbf{Msg}_n) are indexed by $n \in \mathbb{N}$, the number of processes in the protocol. The same is true for the mailboxes ($[\mathbf{Msg}_n]_n$) — a vector of n messages, one for each process. Random access operations for vectors are assumed from Coq’s standard library.	61
FIGURE 4.29	Send and receive events ($E_{\mathbf{net}}$) performed by each process in the leader election protocol. Get and put events ($\mathbf{state}_{\mathbf{PID}_n}$) performed by the round-robin scheduler to track the current running process (\mathbf{PID}_n).	61
FIGURE 4.30	A unidirectional ring with three processes running the leader election protocol. .	62
FIGURE 4.31	Process \mathbf{proc} and a round-robin scheduler \mathbf{rr} implement the leader election protocol. The goal specification is: process $\mathbf{cid} = 3$ <i>eventually</i> receive their own <i>elected</i> message ($\mathbf{E} \ 3$) back.	63
FIGURE 5.1	<i>Zippel</i> protocol between P , V proves the <i>Hadamard</i> product $a * b = c$ in Zero-Knowledge.	67
FIGURE 5.2	Dataflow graph of $\mathbf{hadamard}$ protocol.	68
FIGURE 5.3	Shnorr’s protocol for zero-knowledge identification.	69
FIGURE 5.4	Schnorr’s protocol in <i>Zippel</i> run by \mathcal{P} via the Fiat-Shamir transformation. . .	70
FIGURE 5.5	Recursively sum elements of a vector in <i>Zippel</i> using <i>sized logarithmic recursion</i> . .	77

CHAPTER 1

Introduction

1.1. The Promise and Challenge of Program Verification

In an era where software failures can impact critical infrastructure [Sha21], healthcare [LT93], finance [ZXL20] and military systems [Ste21], formal verification of computer programs provides trust with mathematical certainty. In public-facing systems transparency is essential to establish trust. Users should be able to read a system’s formal specification and use their home computer to check its proof of correctness. Without machine-checked proofs and transparent specifications we must rely on hearsay, reputation and auditing which, while sometimes effective at preventing issues, reduce transparency and accountability.

Despite their promise, adoption of machine-checked proofs in mainstream software development remains challenging. The complexity of software systems often exceeds the expressive power of current methods, while the steep learning curve of verification tools deters developers from integrating these practices. Most critically, verification approaches often lack modularity - without the ability to build proofs incrementally from smaller components, proofs cannot match the pace at which software scales, creating a fundamental barrier to widespread use [CCK⁺20].

This dissertation addresses two fundamental challenges in formal verification: developing expressive specification languages to capture diverse program behaviors, and creating powerful proof systems that enable modular, machine-checked proofs at high levels of abstraction. Leveraging techniques from formal logic and programming languages, we propose several specification languages - from extended regular specifications for finite program traces, to safety and liveness properties for infinite programs with effects and non-determinism, to zero-knowledge properties of cryptographic protocols. For these specification languages, we develop proof systems, model-checking algorithms, and high-level proof lemmas to support practical reasoning. The verification process produces proof certificates that can be readily validated by third parties, providing transparent guarantees of system correctness.

1.2. Expressive formal specifications

A fundamental challenge in verification lies in defining the meaning of correctness. Modern software systems incorporate mutable state, message-passing, randomness, concurrency and non-termination. While programming languages have evolved to express such sophisticated systems compositionally, specification languages often lack the expressiveness to capture desired properties.

Modal logic offers a promising solution. Introducing *modalities* — domain-specific notions such as possibility, necessity, time (temporal logic) and knowledge (epistemic logic) become part of the specification vocabulary. For example, adding the *eventually* operator (\diamond, AF) to a specification language enables reasoning and proving *liveness* properties [Pnu77]. Adding the *done* operator allows reasoning about program termination [DGV⁺13], adding the *knowledge* (K_α) operator enables reasoning about privacy in a multi-user system [Hal97]. Despite their diversity all these notions can be formally defined using simple mathematical models of transition systems, building complex modal lemmas from simple foundations.

In the same vein, *Regular Expressions* (regex) can play the role of specifications describing the behavior of finite program traces. To verify them, many efficient finite automata have been developed. However, the expressive power of regex can be enhanced further; *Extended regular expressions* provide the full power of propositional logic combinators (\wedge, \vee, \neg) alongside traditional regex sequencing operators ($\cdot, *$). Moreover, support for new forms like wildcards ($\cdot, \cdot\{n, m\}$) take this expressiveness even further, allowing us to ignore incosequential parts of a trace and focus on the parts that matter. We build on these observations to define an expressive regular specification language, which is efficient to verify and check.

1.3. Modular machine-checked proofs

The challenge of creating modular, compositional proofs emerges as systems grow in complexity. While individual components may be verifiable in isolation, combining their proofs to reason about larger systems demands powerful abstraction mechanisms. Checking those proofs efficiently remains a challenge.

Model checking provides a foundational framework for modeling diverse computational models, its success in practical verification demonstrates the effectiveness of transition system models [CCK⁺20, Mes08, YML99]. However, traditional model-checking approaches face three significant limitations. First, a verification gap often exists between the abstract model and its concrete implementation in executable code, allowing for potential implementation errors. Second, while transition systems offer mathematical simplicity, they lack natural abstraction mechanisms - reasoning about traces, finite automata, and transitions at a low level impedes structural proofs that reason about high-level program structures like functions, conditionals and loops. Third, most model checking techniques do not produce machine-checked proof certificates that could be independently verified. Program logics like Hoare Logic implemented in proof assistants can address these limitations, but significant effort is required to combine the benefits of both approaches.

1.4. Contributions

This dissertation makes three main contributions to the field of formal verification: First, we present SAFA (Skipping Alternating Finite Automata), a novel automaton model based on Alternating Finite Automata (AFA) [CKS81] that models the language of *Extended Regular Expressions with wildcards* — irrelevant regions of the input trace that can be safely skipped. SAFA’s unique design enables a multithreaded verification algorithm and produces compact proof objects that can be efficiently checked. The combination of fast verification and small proofs makes SAFA particularly well-suited for the Reef [AIM⁺24] compiler for zero-knowledge cryptographic certificates of regex matching over secret documents.

Second, we graduate from finite programs with regular language traces to potentially infinite programs with effects, ghost-state, nondeterminism and concurrency, and from model-checking techniques to structural proofs in the Coq proof assistant [The24]. We introduce Tiel (Temporal Interaction and Choice Logic), a structural temporal logic that bridges the gap between program logics, such as Hoare Logic, and temporal logics, such as LTL and CTL [Pnu77, EC82]. Tiel internalizes complex (co-)inductive proof techniques to structural lemmas and reasoning about variants and invariants. We show that it is possible to perform mechanized proofs of (nested) safety and liveness

properties, while working in a high-level of abstraction. Ticl supports various programming languages and side-effects by using a programming model in the Interaction Trees family [XZH⁺20]. We demonstrate Ticl’s expressiveness through a series of case studies drawn from computer systems — a round-robin task scheduling service, concurrent shared memory systems, and a distributed consensus protocol.

Third, we propose Zippel, a programming language designed for implementing zero-knowledge cryptographic proof systems with strong safety guarantees. Zippel’s strict type system ensures safety from simple errors, while a static analysis based on Information Flow security ensures no information is leaked to unprivileged parties. The goal is to formally show well-typed Zippel protocols are zero-knowledge. In addition, Zippel has a compiler that can produce fast, multithreaded implementations from high-level cryptographic protocol descriptions. Zippel’s approach to zero-knowledge protocols bridges the gap between theoretical security properties and practical cryptographic systems.

1.5. Dissertation Overview

This dissertation is organized as follows:

Chapter 2 provides the necessary background on automata theory, regular expressions, modal logic, temporal logic, and epistemic logic needed to understand our technical contributions. We review key concepts from formal verification such as safety and liveness properties, and discuss the state of the art in program logics and liveness property verification.

Chapter 3 introduces SAFA and the specification language of extended regular expressions. We describe the implementation of the SAFA compiler based on regular expression derivatives [Brz64], and prove the translation from Regular Expressions to SAFA is sound. We then describe an efficient algorithm that verifies a trace belongs in a language using SAFA, producing succinct proof objects.

Chapter 4 presents Ticl and its implementation in the Coq proof assistant [The24]. We detail the semantics of Interaction and Choice Trees (ictrees) a denotational model of programming languages with effects, nondeterminism and non-termination. We give a small-step semantics to ictrees via a Kripke semantics and define the syntax and semantics of Ticl. We prove a multitude of forward

and backwards structural reasoning lemmas, that internalize (co-)inductive techniques to reasoning about variants, invariants and postconditions. Through case studies of languages for job servers, concurrent memory systems, and distributed consensus protocols, we demonstrate how Ticl enables small, structural and composable proofs in very few lines.

Chapter 5 motivates and proposes Zippel, a strictly-typed programming language designed for zero-knowledge cryptographic proof systems. We define the type system for Zippel, using sized-types, and formalize the language semantics. Finally, we propose a novel application of epistemic logic to zero-knowledge reasoning via static analysis of the Zippel transcript. We plan to demonstrate the language through several cryptographic protocol implementations, such as Schnorr’s protocol [Sch91b], Marlin [CHM⁺20], Nova [KST22a] and Groth16 [Gro16].

Chapter 6 concludes by summarizing our contributions and discussing future directions for research in automated verification of distributed systems and cryptographic protocols.

CHAPTER 2

Background and related work

This chapter reviews necessary background on modal logic, Kripke structures, finite automata, regular languages, temporal logic, and epistemic logic, needed to understand our technical contributions. We review key concepts from formal verification such as safety and liveness properties, and discuss the state of the art in program logics and the connection between epistemic logic and zero-knowledge.

2.1. Modal Logic and Kripke Structures

Modal logic extends classical propositional logic by introducing operators that qualify the truth of statements in different contexts or “possible worlds”. While classical logic deals with absolute truths, modal logic allows us to reason about necessity, possibility, and other modalities that arise naturally when analyzing computer systems. For instance, when verifying a concurrent program, we might want to express properties like “eventually the program terminates” or “the mutual exclusion invariant is always maintained”.

At the heart of modal logic’s semantics lies the concept of Kripke structures, named after Saul Kripke’s seminal work in the 1950s. A Kripke structure $M = (W, R, L)$ consists of:

- A non-empty set W of worlds (or states).
- A binary relation $R \subseteq W \times W$ called the accessibility relation.
- A labeling function $L : W \rightarrow 2^{AP}$ that assigns to each world the set of atomic propositions true in that world.

In the context of program verification, worlds often represent program states, the accessibility relation captures possible transitions between states, and the labeling function indicates which program properties hold in each state. For example, in a simple mutual exclusion protocol, a world might represent a state where “process 1 is in its critical section” and the accessibility relation would capture how processes can move between their critical and non-critical sections.

The syntax of modal logic formulas differs, take for example a modal logic of “necessity” and “possibility” with syntax given by the following grammar:

$$\langle \varphi \rangle ::= \perp \mid \top \mid \neg \varphi \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \varphi \rightarrow \varphi' \mid \Box \varphi \mid \Diamond \varphi$$

Where p ranges over a set AP of atomic propositions. The *box* operator \Box represents necessity or universality— $\Box\phi$ means that ϕ holds in all accessible worlds from the current state. The *diamond* operator \Diamond represents possibility or existence— $\Diamond\phi$ means that ϕ holds in at least one accessible world from the current state. These operators are duals, meaning $\Diamond\phi \equiv \neg\Box\neg\phi$. For example, in a program verification context, $\Box\phi$ might mean “ ϕ holds after every possible program step” while $\Diamond\phi$ means “ ϕ holds after some possible program step”.

Formulas in modal logic are interpreted through the satisfaction relation $M, w \models \phi$. For world w and structure M , satisfaction is defined inductively on the structure of formulas φ :

$$\begin{array}{ll} M, w \models p & \text{iff } p \in L(w) \text{ for atomic propositions } p \\ M, w \models \neg\phi & \text{iff } M, w \not\models \phi \\ M, w \models \phi \wedge \psi & \text{iff } M, w \models \phi \text{ and } M, w \models \psi \\ M, w \models \Box\phi & \text{iff } \forall v \in W : (w, v) \in R \implies M, v \models \phi \\ M, w \models \Diamond\phi & \text{iff } \exists v \in W : (w, v) \in R \text{ and } M, v \models \phi \end{array}$$

Different properties of the accessibility relation R give rise to different modal logics, each capturing distinct reasoning principles. The most important properties include:

- Reflexivity: $\forall w \in W : (w, w) \in R$
- Transitivity: $\forall w, v, u \in W : (w, v) \in R \text{ and } (v, u) \in R \implies (w, u) \in R$
- Symmetry: $\forall w, v \in W : (w, v) \in R \implies (v, w) \in R$
- Totality (left): $\forall w \in W : \exists v \in W : (w, v) \in R$

These properties correspond to important axioms in modal logic:

- Reflexivity yields axiom T: $\Box\phi \implies \phi$
(“if something is necessary, then it is true”)
- Transitivity yields axiom 4: $\Box\phi \implies \Box\Box\phi$
(“if something is necessary, then it is necessarily necessary”)
- Symmetry yields axiom B: $\phi \implies \Box\Diamond\phi$
(“if something is true, then it is necessarily possibly true”)
- Left-totality yields axiom D: $\Box\phi \implies \Diamond\phi$
(“if something is necessary, then it is possible”)

Different combinations of these properties give rise to well-known modal logics:

- System K: The basic modal logic with no special properties.
- System T: Adds reflexivity to K.
- System S4: Adds both reflexivity and transitivity to K.
- System S5: Adds reflexivity, transitivity, and symmetry to K.

These systems have found various applications in computer science. S4 is often used in epistemic logic, where reflexivity captures the truth axiom (if an agent knows ϕ , then ϕ is true) and transitivity captures positive introspection (if an agent knows ϕ , they know that they know it). S5 adds negative introspection (if an agent doesn’t know ϕ , they know that they don’t know it). Finally, totality is important in temporal logic, ensuring that time always progresses.

The correspondence between frame properties and axioms is not merely syntactic—it reflects deep connections between the algebraic properties of accessibility relations and the logical principles they support. This correspondence theory helps us choose the appropriate modal logic for specific applications by matching the desired reasoning principles with the corresponding structural properties.

2.2. Finite automata

We review some introductory automata theory, this will be useful background for regex compilation and the SAFA definition in Section 3.4. We define here *Deterministic Finite Automata* (DFA), *Nondeterministic Finite Automata* (NFA) and *Alternating Finite Automata* (AFA) introduced by Dexter Kozen in his seminal work on *Alteration* [CKS81]. Both DFA, NFA and AFA are mathe-

mathematical structures based on a finite set of states Q and a transition relation δ that describes the behavior of the automaton from state to state.

Deterministic Finite Automata (DFA) are formally defined in Figure 2.1. DFA start at an initial state q_0 and transition to subsequent states in Q , using the transition relation δ to determine the next step.

Definition 2.2.1. A DFA is a 5-tuple $\mathcal{N} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ where

Q	:	the set of all states
$\Sigma \cup \{\epsilon\}$:	The alphabet
$q_0 \in Q$:	Initial state
$\delta \subseteq Q \times \Sigma \times Q$:	Transition relation
$\mathcal{F} \subseteq Q$:	Set of accepting states

Figure 2.1: Deterministic Finite Automata (DFA) have a deterministic transition relation.

Nondeterministic Finite Automata (NFA) are formally defined in Figure 2.2. NFA start at an initial state q_0 and transition *nondeterministically* to subsequent states in Q , using the transition relation δ to determine the next step. Note the special ϵ nondeterministic transition, which does not consume a character and allows the NFA to arbitrarily (nondeterministically) take those transitions.

Definition 2.2.2. A NFA is a 5-tuple $(Q, \Sigma, q_0, \delta, \mathcal{F})$ where

Q	:	the set of all states
$\Sigma \cup \{\epsilon\}$:	The alphabet with ϵ (empty string)
$q_0 \in Q$:	Initial state
$\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$:	Transition relation
$\mathcal{F} \subseteq Q$:	Set of accepting states

Figure 2.2: Noneterministic Finite Automata (NFA) can chose ϵ transitions.

AFA [CKS81] are formally defined in Figure 2.3 — AFA have two kinds of state, existential (\exists) and universal (\forall). When a state is an \exists state, based on the labeling function λ_q , the transition is

nondeterministic — given string s any one transition that leads to an accepting state for s can be chosen. Dually, a \forall state accepts if *all* descendants of the state lead to an accepting state for s .

Definition 2.2.3. An AFA [CKS81] is a 6-tuple $(Q, \Sigma, \lambda_q, \delta, q_0, \mathcal{F})$ where

Q	:	the set of all states
Σ	:	The alphabet
$q_0 \in Q$:	Initial state
$\lambda_q : Q \rightarrow \{\forall, \exists\}$:	Label states \forall or \exists
$\delta \subseteq Q \times \Sigma \times Q$:	Transition relation
$\mathcal{F} \subseteq Q$:	Set of accepting states

Figure 2.3: An AFA alternates between existential and universal states.

2.3. Extended Regular Expressions

We introduce the extended syntax of regular expressions (regex) in Figure 2.4. Extended regex syntax includes conjunction (\wedge) and negation (\neg), important for writing expressive specifications. We additionally add bounded repetition to this syntax with no change the regular nature of the source language, as $r\{a, b\} = ra \vee ra + 1 \vee \dots \vee rb$ for $a \leq b$, $r\{0\} = \epsilon$ and $r\{a\} = r \cdot r\{a - 1\}$ for $a > 0$. The regex syntax in Figure 2.4 is the same as Owens et al [ORT09] with the addition of *bounded repetition*.

$r, s ::= \perp$	Empty regex
ϵ	Empty string
\mathcal{C}	Non-empty character set ($\emptyset \subset \mathcal{C} \subseteq \Sigma$)
$r \cdot s$	concatenation
$r \vee s$	Choice (disjunction)
$r \wedge s$	Logical and (conjunction)
$\neg r$	Logical negation
r^*	Kleene-closure
$r\{n, m\}$	Bounded repetition ($n, m \in \mathbb{N}$)

Figure 2.4: Regular expression (regex) syntax. Additional syntactic notations are the wildcard \cdot defined as the character set Σ , n -repetition $r\{n\}$ as $r\{n, n\}$ and the match-all \top as \cdot^* . We may omit the concatenation operator when it is implied.

Now let's define the *language* of a regular expression r ; given an alphabet Σ , the regular language accepted by regex r is defined as $\mathcal{L}[\llbracket r \rrbracket] \subseteq \Sigma^*$ in Figure 2.5). Regular languages are recognized by

finite automata and every regular language has a corresponding finite automaton and vice versa via the Myhill-Nerode theorem.

With regards to efficient computation of trace membership in a language $S \in \mathcal{L}[[r]]$, it is not possible to use the language definition in Figure 2.5 as an algorithm because languages are infinite, for example $\mathbf{a^*}$. Even if we restrict $\mathcal{L}[[r]]$ to the finite set of all S -length traces, enumerating $\mathcal{L}[[r]]$ can be very computationally expensive – we have to enumerate a potentially all the traces even after encountering $S \in \mathcal{L}[[r]]$. From a computational perspective it is much better to compile regular expression r to a finite automaton and use that to do the matching.

Definition 2.3.1.

$$\begin{aligned}
\mathcal{L}[[\emptyset]] &= \emptyset \\
\mathcal{L}[[\epsilon]] &= \{\epsilon\} \\
\mathcal{L}[[C]] &= \mathcal{C} \quad (\emptyset \subset C \subseteq \Sigma) \\
\mathcal{L}[[r \cdot s]] &= \{uv \mid u \in \mathcal{L}[[r]], v \in \mathcal{L}[[s]]\} \\
\mathcal{L}[[r \vee s]] &= \mathcal{L}[[r]] \cup \mathcal{L}[[s]] \\
\mathcal{L}[[r \wedge s]] &= \mathcal{L}[[r]] \cap \mathcal{L}[[s]] \\
\mathcal{L}[[r^*]] &= \{\epsilon\} \cup \mathcal{L}[[r \cdot r^*]] \\
\mathcal{L}[[\neg r]] &= \Sigma^* \setminus \mathcal{L}[[r]] \\
\mathcal{L}[[r\{n, m\}]] &= \begin{cases} \{\epsilon\} & \text{if } n = m = 0 \\ \mathcal{L}[[r \cdot r\{0, m-1\}]] & \text{else if } n = 0 \\ \mathcal{L}[[r \cdot r\{n-1, m-1\}]] & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 2.5: The set of strings accepted by a regex r is the *language* $\mathcal{L}[[r]] \subseteq \Sigma^*$.

Next, we introduce *regex derivatives* as a way to compile regex to finite automata. First introduced by Brzozowski[Brz64], regex derivatives give a simple regex compiler implementation (by comparison to Thompson’s NFA compiler [Tho68]), while maintaining a low number of states in the result FA. Consequent works improve on that number [Ant96, ORT09, CCM14].

2.4. Derivatives of regular expressions

Brzozowski[Brz64] defined the derivative of a regex r given a character $\alpha \in \Sigma$ as $d_\alpha(r)$, another regular expression such that its language $\mathcal{L}[[d_\alpha(r)]]$ contains all the suffixes $w \in \Sigma^*$ in $\mathcal{L}[[r]]$ that

have an α prefix.

$$\mathcal{L}[\llbracket r \rrbracket] = \{\alpha w \mid w \in \mathcal{L}[\llbracket d_\alpha(r) \rrbracket]\}$$

Regex derivatives are then used to compile regular expressions to deterministic finite automata (DFA), following the recursive algorithm in Figure 1.

We must first define the *nullable* predicate $v(r)$ in Figure 2.6 — $v(r)$ is true if and only if the regex r accepts the empty string. Nullable regex correspond to *accepting* states in finite automata, the intuition is that accepting the empty string indicates it is okay to stop consuming characters. We use $v(r)$ in Definitions 2.7, 3.4 to check if for a regex $r \cdot s$ the derivative $d_\alpha(r \cdot s)$ should be applied to r or s .

Definition 2.4.1.

$$\begin{aligned} v(\epsilon) &= true \\ v(r^*) &= true \\ v(\perp) &= false \\ v(\mathcal{C}) &= false \\ v(r \cdot s) &= v(r) \wedge v(s) \\ v(r \vee s) &= v(r) \vee v(s) \\ v(r \wedge s) &= v(r) \wedge v(s) \\ v(r\{n, m\}) &= \begin{cases} true & \text{if } n = 0 \\ v(r) & \text{otherwise} \end{cases} \end{aligned}$$

Figure 2.6: The predicate $v(r)$ is true when the regex r accepts the empty string.

Now the Brzozowski derivative [Brz64, ORT09] $d_\alpha(r)$ for a character $a \in \Sigma$ and regex r is another regex, defined recursively in Figure 2.7.

2.4.1. Regex compiler to DFA

Now proceed to the DFA construction method using Brzozowski derivatives. DFA construction is implemented as a simple worklist algorithm in Figure 1.

Note the algorithm in Figure 1 checks if a regex is included in the set of already visited states. A

Definition 2.4.2.

$$\begin{aligned}
d_\alpha(\perp) &= \emptyset \\
d_\alpha(\epsilon) &= \emptyset \\
d_\alpha(\mathcal{C}) &= \begin{cases} \epsilon & \text{if } \alpha \in \mathcal{C} \\ \perp & \text{otherwise} \end{cases} \\
d_\alpha(r \cdot s) &= \begin{cases} (d_\alpha(r)s) \vee d_\alpha(s) & \text{if } v(r) \\ d_\alpha(r)s & \text{otherwise} \end{cases} \\
d_\alpha(r \vee s) &= d_\alpha(r) \vee d_\alpha(s) \\
d_\alpha(r \wedge s) &= d_\alpha(r) \wedge d_\alpha(s) \\
d_\alpha(\neg r) &= \neg d_\alpha(r) \\
d_\alpha(r^*) &= d_\alpha(r)r^* \\
d_\alpha(r\{n, m\}) &= \begin{cases} \emptyset & \text{if } n = m = 0 \\ d_\alpha(r \cdot r\{0, m-1\}) & \text{if } n = 0 \\ d_\alpha(r \cdot r\{n-1, m-1\}) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 2.7: Brzowski derivative for a regex r given character $\alpha \in \Sigma$ is $d_\alpha(r)$.

naive equality check would lead to a blow-up in the number of states [Brz64], considering *equivalent* states separately. For example the regex $\epsilon \cdot r \neq r$ but the two regex are equivalent as ϵ is the identity element of concatenation. Instead, a weaker notion of syntactic regular expression equivalence is required, one that approximates language equivalence such that $r \simeq s \iff \mathcal{L}[[r]] = \mathcal{L}[[s]]$.

2.4.2. Weak regex equivalence

Owens et al defines a weaker notion of equivalence which significantly reduces the number of states in the DFA [ORT09]. This syntactic equivalence encodes properties like commutativity, associativity and unit for conjunction (\wedge), disjunction (\vee) and associativity and unit for concatenation (\cdot). However, equality checking under a set of rewriting rules is computationally expensive, creating a trade-off between slow equivalence checking or a large number of states.

An alternative is using Hopcroft's algorithm [Hop71] for DFA minimization. We revisit the state explosion problem in Section 3.4 and show a generalization of Antimirov's algorithm for partial regex derivatives [Ant96] is effective in avoiding state explosion without sacrificing expressivity.

Algorithm 1: Brzozowski's regex compiler to DFA using derivatives.

Input: Regular expression r

Output: DFA $D = (Q, \Sigma, \delta, q_0, \mathcal{F})$

$Q \leftarrow \emptyset;$

$WorkList \leftarrow \{r\};$

$q_0 \leftarrow r;$

// Initial state corresponds to regex r

$\mathcal{F} \leftarrow \emptyset;$

$\delta \leftarrow \emptyset;$

while $WorkList \neq \emptyset$ **do**

$r_c \leftarrow WorkList.pop();$

$Q \leftarrow Q \cup \{r_c\};$

if $v(r) = true$ **then**

$\mathcal{F} \leftarrow \mathcal{F} \cup \{r_c\};$

// Nullable regex correspond to accepting states

end

forall $a \in \Sigma$ **do**

$r_n \leftarrow d_a(r_c);$

// Compute regex Derivative for next state

$\delta \leftarrow \delta \cup \{(r_c, a, r_n)\};$

if $r_n \notin Q$ **then**

$WorkList \leftarrow WorkList \cup \{r_n\};$

end

end

end

CHAPTER 3

Skipping Alternating Finite Automata (SAFA)

3.1. Regular expression specifications

Regular expressions (regex) are widely used for text search operations in tools like **grep**. Here, we explore an alternative application: using regular expressions as a specification language for finite linear traces. Specifically, we focus on *extended regular expressions* (Section 2.3), which augment standard regex operations of disjunction (\vee), concatenation (\cdot) and closure ($*$) with two additional logical operators: conjunction (\wedge) and negation (\neg). This extension elevates the expressive power to match that of propositional logic over finite traces.

Verifying compliance with regular specifications can be computationally intensive, particularly when handling specifications with nondeterministic elements ($r \vee s$, r^*) over long traces σ . We address this performance challenge through two approaches. First, we develop a compiler that transforms regular specifications into efficient automata while minimizing state count. Second, we introduce a system for generating compact *proof objects* that act as verification “receipts”. This approach produces a concise certificate π that can confirm trace $\sigma \in \mathcal{L}[[r]]$ more efficiently than repeating the full verification of regex r . To achieve these goals, we introduce SAFA (Skipping Alternating Finite Automata), a novel class of finite automata. SAFA effectively handles the state explosion problem inherent to extended regex—caused by the distributivity of conjunction and sequencing operators (Section 2.4.1)—while identifying the minimal set of characters in a trace necessary for proof checking. We begin with a review of foundational automata theory and extended regex syntax and semantics in Sections 2.2 and 2.3. Building on these concepts, we then present SAFA, which extends Alternating Finite Automata (AFA) by introducing *skips*—regions of the trace that can be safely ignored during verification.

3.2. Alternating Finite Automata (AFA)

AFA [CKS81] are finite automata that generalize *Nondeterministic Finite Automata* (NFA), labeling states with an existential (\exists) or a universal (\forall) quantifier. An \exists state is identical to a state in an

NFA; the AFA merely reads the character at the current cursor, advances the cursor, and then transitions to any one of its possible next states. A \forall state is very different. First, the AFA creates a *copy* of the remaining characters in the input string (starting at the current cursor until the end of the string) for *each* of its transitions (i.e., if there are 10 transitions it will create 10 copies of the input string). Then, in parallel, it transitions to every next state, and feeds each of those states their own independent copy of the input. For the AFA to accept an input string, all of the parallel branches need to end in accepting states. Intuitively, \forall states capture the conjunction of multiple sub-automata, each of which operates independently on the provided input. AFA are defined formally in Figure 2.3.

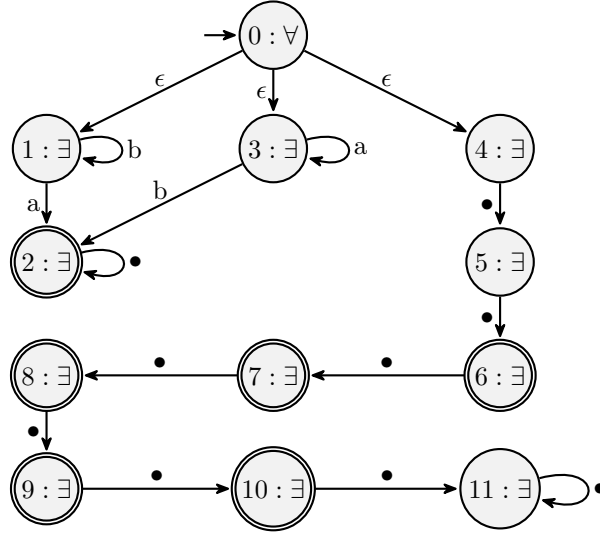


Figure 3.1: AFA for regex $r = (. * a) \wedge (. * b) \wedge (. \{2, 6\})$.

Example. Suppose we want to match traces of length between 2–6 that contain “a” and “b” defined over alphabet $\Sigma = \{a, b, c\}$. This is given by the regex $r = “(. * a) \wedge (. * b) . \{2, 6\}”$ (note we may omit the concatenation operator when it is implied). Representing r as an NFA requires creating an automaton that accepts the alternation of all strings that contain both “a” and “b” and have length between 2 to 6 (“ab”, “.ab”, “a..b”, “.a.b”, etc.). The minimal NFA for this has 18 states, but the corresponding AFA has 11-states, given in Figure 3.1.

To understand this AFA, first recall *epsilon transitions*, which AFA inherit from NFA and which

mean that the automaton can take any transition with an ϵ label without advancing the cursor or reading any character from the trace. Second, notice the state at the top is labeled \forall , which means that after processing the trace, all of its transitions (the 3 vertical branches) should end in an accepting state. The transitions of \forall states are special in that each creates a private copy of the cursor initialized to the value of the cursor when the \forall state is reached. As a result, states 1, 3, and 5 will all have their own cursors (i.e., advancing the cursor of the left branch does not affect the cursor of the right branch).

Consider for example the trace $\sigma = "acbcc"$ which is accepted since the three branches out of state 0 run in parallel and each branch terminates in an accepting state. If instead $\sigma = "bccbb"$, the middle and right branches both terminate in accepting states, but the left branch does not.

The above example immediately shows that AFA could provide savings over the automata considered by prior works. Indeed, if a regex requires n states to be represented in an AFA, the same regex may require 2^{2^n} states in a DFA [FJY90].

3.3. Supporting Skips

AFA are a great way to increase the expressiveness of the supported regular expressions without incurring exponential costs — still proof objects are linear with respect to the trace. As a concrete example, consider the regex $r = ".^*ab"$ and the trace $\sigma = "aaab"$. AFA (much like NFA) represent $".^*" by a single, non-accepting state, with the option to loop or progress forward with an ϵ transition. Finding the solution to the question “is $\sigma \in \mathcal{L}[[r]]$ ”? requires computing both the case in which the first “a” in σ matches the “.” in r and the case in which it matches the “a” in r . Confirming a match is simpler: given a path through the AFA for σ , we just need to check that the path leads to an accepting state.$

We can even take this concept further. When computing, bounded wildcard matching has to be explicitly unrolled. $".\{m,n\}"$ and $".\{n\}"$ require at least n transitions in an NFA or AFA. We see this in the right branch of the AFA in Figure 3.1 (states 4 through 10), where each state in $".\{2,6\}"$ has to be included explicitly.

But when checking, what if we could simply move the cursor forward by a number between 2–6 (inclusive), and carry on? Since “.{2,6}” is a wildcard, the content does not matter; what matters is that a *wildcard region* of the appropriate length exists. To express wildcard regions, we introduce *skips*. A skip is a disjoint set datastructure [ST81] composed of non-overlapping intervals, $s = \{i_1, \dots, i_n\}$, where each interval is of the form $i = [start, end]$ or $i = [start, \infty)$. Both *start* and *end* are natural numbers with $start \leq end$; for $[start, \infty)$, the interval is unbounded on the right-hand side indicating it can span to the end of any trace.

The properties of skips are; succinct representation of (infinite) sets of natural numbers, fast membership checks, and closure with respect to concatenation (\cdot), Kleene-star ($*$) and bounded repetition ($\{n, m\}$). For example in the regex $\{2, 3\}.a$ two adjacent wildcards compose and we get the equivalent skip $\{3, 4\}$. Negation of a skip gives the complement set, for example $\neg\{3, 4\}$ gives us the disjoint set $\{[0, 2], [5, \infty)\}$.

The idea is that when we reach a state that has a *skip transition* s , instead of reading a character from the input and transitioning to the next state based on the read value, the automaton advances the cursor by any amount within the intervals in s , and then moves to the next state. An implicit assumption is random, forward access to the input trace σ , meaning we can fast-forward any number of positions, but not rewind to a previous position.

Also, observe that skips generalize epsilon transitions: we can simply define skip $\epsilon = \{[0, 0]\}$. Third, we can support Kleene-star wildcard regions with skip $* = \{[0, \infty)\}$, meaning any cursor less-than the length of the trace works. The use of the ∞ symbol allows the separation of SAFA from the trace to which it is applied. Interval sets are also sets of natural numbers so we use set notation $i \in s$ to indicate membership of natural number i in any of the disjoint intervals in the skip, which we can compute in time sublinear to the number of intervals.

3.3.1. Intervals and Interval Sets (skips)

We formally define *intervals* and *interval sets* (skips) before moving to defining SAFA.

A *bounded interval* $[a, b]$ where $a \leq b$, $a, b \in \mathbb{N}$ represents the subset of natural numbers $\{i \mid a \leq i \leq$

$b\}$, inclusive in both ends. An *unbounded interval* $[a, \infty)$ represents the subset of natural numbers $\{i \mid a \leq i\}$, inclusive on the left, unbounded on the right. We represent intervals with the letter i .

An *interval set* S or a *skip* is a disjoint set of intervals $\{i_1, \dots, i_n\}$ ordered by increasing starting point, such that there is no *overlap* between consecutive intervals.

Interval sets allow us to represent continuous ranges of natural numbers, for example the single interval $\{[a, b]\}$ where $a \leq b$, $a, b \in \mathbb{N}$, as well as non-continuous intervals, for example the set of numbers less-than-equal to a and greater-than-equal b as $\{[0, a], [b, \infty)\}$, assuming $a < b$. A right-open interval simply means it can span until the end of the trace.

We consider an overlap of two intervals to be a difference of at most 1 between the end-point of the first and the start point of the second. So for example $[1, 2], [4, 5]$ do not overlap but $[1, 3], [2, 4]$ and $[1, 2], [3, 4]$ overlap and are both equivalent to $[1, 4]$. Two intervals $[a_1, b_1], [a_2, b_2]$ overlap if $\max(a_1, b_1) + 1 \geq \min(a_2, b_2)$.

3.3.2. Operations on Interval sets

Interval sets admit the familiar *union*, *concatenation*, *Kleene-star* and *bounded repetition* operations from regex, defined in Figure 3.5. We use set membership notation $i \in s$ and $n \in s$, to mean i is an interval in s , or n is a number contained in one of the intervals of s .

With interval sets defined proceed to formally define a new finite automaton that takes advantage of the skip structure to “fast-forward” over irrelevant areas of the trace.

3.4. SAFA formal definition

A Skipping Alternating Finite Automaton (SAFA) is an 8-tuple $(Q, E, \Sigma, \lambda_q, \lambda_e, \delta, q_0, \mathcal{F})$ such that

Q	: the set of all states (nodes)
E	: the set of all transitions (edges)
Σ	: The alphabet
$\lambda_q : Q \rightarrow \{\forall, \exists\}$: Label nodes, \forall or \exists
$\lambda_e : E \rightarrow S \uplus C$: Label edges, skip or character set
$\delta \subseteq Q \times E \times Q$: Transition relation
$q_0 \in Q$: Initial state
$\mathcal{F} \subseteq Q$: Set of accepting states

SAFA are similar to AFA with the addition of an edge labeling function λ_e . λ_e can be thought of as an analog of λ_q , but over transitions instead of states. Transitions in a SAFA can be labeled as either a character set $C \subseteq \Sigma$ which match a character $\sigma_i \in C$, for trace σ , or a skip s , which does not consume a character, but increases the cursor nondeterministically by any $n \in s$.

We overload the notation $(q, s, q') \in \delta$ to indicate a skip transition s , or more precisely there exists an edge $e \in E$ such that $(q, e, q') \in \delta$ and $\lambda_e(e) = s$. Also overload $(q, \alpha, q') \in \delta$ to indicate a character $\alpha \in \Sigma$ transition, or there exists an edge $e \in E$ and character set $C \subseteq \Sigma$ such that $(q, e, q') \in \delta$ and $\lambda_e(e) = C$ and $\alpha \in C$. As the labeling function λ_e maps to a disjoint union there is no confusion with this notation.

Example. In Figure 3.2 we show the SAFA that corresponds to the AFA from Figure 3.1.

The SAFA replaces the long chain of states (4–10) in the AFA with $Skip\{[2, 6]\}$. This compression is possible because ϵ (the identity element) followed by skip s is just s . The examples in Figures 3.1 and 3.2 provide the intuition for why SAFA has smaller proof objects than AFA, while also being computationally equivalent. We formalize the equivalence between SAFA and AFA by direct translation.

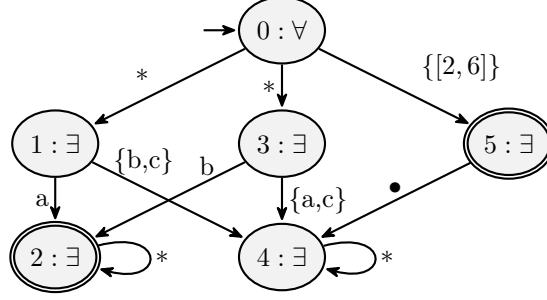


Figure 3.2: SAFA for regex $r = (. * a) \wedge (. * b) \wedge (. \{2, 6\})$ over alphabet $\Sigma = \{a, b, c\}$ — edges $*$ mean the skip $\{[0, +\infty)\}$.

Theorem 3.4.1. *Let \mathcal{S} denote a SAFA. There exists an AFA \mathcal{A} such that the language $\mathcal{L}[\![\mathcal{S}]\!] = \mathcal{L}[\![\mathcal{A}]\!]$ and $\mathcal{L}[\![\mathcal{A}]\!]$ is regular.*

Proof: The proof is by translating a SAFA to an AFA by expanding skips to equivalent AFA. Every skip is an interval set with finite intervals $\mathcal{S} = \{i_0, \dots, i_n\}$ and for every finite interval $i_n = [a, b] = .\{a, b\}$ there exists an AFA with $b - a$ states that recognizes it. For every right-open interval $i_n = [a, \infty)$ there exists an AFA with $a + 1$ states, where the last state is a self-looping, accepting state that recognizes it.

3.5. SAFA compiler

$$\begin{array}{c}
\frac{}{\text{take}_s(.) = (\{[1, 1]\}, \epsilon)} \text{DOT} \qquad \frac{\text{take}_s(r) = (s, \epsilon) \quad s \neq \{\}}{\text{take}_s(r^*) = (s^*, \epsilon)} \text{STAR}^* \\
\\
\frac{\text{take}_s(r) = (s, \epsilon) \quad s \neq \{\}}{\text{take}_s(r\{i, j\}) = (s\{i, j\}, \epsilon)} \text{REP} \qquad \frac{\text{take}_s(r_1) = (s_1, \epsilon) \quad \text{take}_s(r_2) = (s_2, r')}{\text{take}_s(r_1 \cdot r_2) = (s_1 \cdot s_2, r')} \text{APPR} \\
\\
\frac{\text{take}_s(r_1) = (s, r')}{\text{take}_s(r_1 \cdot r_2) = (s, r'_1 \cdot r_2)} \text{APPL}
\end{array}$$

Figure 3.3: Rules for a partial, recursive function $\text{take}_s(r) = (s, r')$ extracting skip s from the head a regex r and returning the tail r' .

We present a compiler from extended regex to SAFA. As SAFA support skipping over wildcard regions, define the rules for extracting skips from the beginning of regex one at a time, with the partial function $\text{take}_s(r) = (s, r')$ that extracts skip s (Figure 3.3).

3.5.1. Generalized Antimirov Partial Derivatives

Background Section 2.2 gives a primer on finite automata and regular expression (regex) compilers based on Brzozowski regex derivatives [Brz64, ORT09]. Owens et al [ORT09] improves on the state explosion problem (Section 2.4.1) by using a weaker notion of approximate regex equivalence. Weak regex equivalence incorporates algebraic properties of regex such as commutativity, associativity, identity, annihilation and more, in order to syntactically approximate language equivalence, or $r \simeq s \iff \mathcal{L}[\![r]\!] = \mathcal{L}[\![s]\!]$. In reality, equational reasoning may require a large number of rewrites to show equivalence of two regular expressions. It may seem we have traded unbounded DFA states for unbounded time to check for regex equivalence, but this is not the end of the road.

Antimirov improves on regex derivatives with *Partial Regex derivatives* [Ant96] by observing that *sets of regex* form a semilattice with respect to set union and the empty set (R, \cup, \emptyset) , such that regex associativity, commutativity, and zero-element laws are obtained from the set structure instead of being encoded via regex equivalence. Unfortunately, Antimirov’s technique does not apply to extended regex with conjunction (\wedge) and negation (\neg). Caron et al. [CCM14] generalize Antimirov’s partial derivatives from sets to arbitrary *support structures*. We choose the free positive boolean algebra $B^+(\text{regex})$ as our support structure, meaning any boolean expression over regex with no negation (\neg) (Appendix A.1.2). Then define the *Generalized Antimirov partial derivatives* $\partial_\alpha(r) : B^+(\text{regex})$ such that, given a character $\alpha \in \Sigma$ and a regex r , $\partial_\alpha(r)$ returns a free *positive* boolean algebra of regex $B^+(\text{regex})$.

In practice, the free positive boolean algebra $B^+(\text{regex})$ is simply a set of sets, of regex, encoding the Disjunctive Normal Form (DNF) of regex terms. The outer set represents a logical disjunction (\vee) while the inner sets a logical conjunction (\wedge). Associativity and commutativity are obtain by the semilattice structure of the nested sets, similar to Antimirov’s derivatives [Ant96]. Distributivity

is encoded syntactically, by expanding the inner set into the outer set, for example

$$\begin{aligned}
& \partial_\alpha(r \& (s + t)) \\
&= \partial_\alpha(r) \wedge \partial_\alpha(s + t) \\
&= (\partial_\alpha(r) \wedge \partial_\alpha(s)) \vee (\partial_\alpha(r) \wedge \partial_\alpha(t))
\end{aligned}$$

While distributivity can cause a state explosion if there are many nested conjunctions, based on our experiments with regular expressions we discovered there are not many deeply nested conjunctions and we were able to get a small number of states.

Here's our definition of generalized partial derivative $\partial_\alpha(r)$ given character $\alpha \in \Sigma$ in Figure 3.4.

Definition 3.5.1.

$$\begin{aligned}
\partial_\alpha(\perp) &= \perp \\
\partial_\alpha(\epsilon) &= \perp \\
\partial_\alpha(\mathcal{C}) &= \begin{cases} \epsilon & \text{if } \alpha \in \mathcal{C} \\ \perp & \text{otherwise} \end{cases} \\
\partial_\alpha(r \cdot s) &= \begin{cases} \partial_\alpha(s) \vee \partial_\alpha(r) \cdot s & \text{if } v(r) \\ \partial_\alpha(r) \cdot s & \text{otherwise} \end{cases} \\
\partial_\alpha(r \vee s) &= \partial_\alpha(r) \vee \partial_\alpha(s) \\
\partial_\alpha(r \wedge s) &= \partial_\alpha(r) \wedge \partial_\alpha(s) \\
\partial_\alpha(r^*) &= \partial_\alpha(r) \cdot r^* \\
\partial_\alpha(r\{n, m\}) &= \begin{cases} \perp & \text{if } n = m = 0 \\ \partial_\alpha(r \cdot r\{0, m-1\}) & \text{if } n = 0 \\ \partial_\alpha(r \cdot r\{n-1, m-1\}) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.4: The generalized partial regex derivative $\partial_\alpha(r)$ is a positive boolean (DNF) expression over regex.

As every generalized Antimirov partial derivative will return a set of sets, a compiler from Extended Regex to AFA will instantiate for every regex, one level of existential (\exists) states (corresponding to outer disjunction), followed by another level of universal (\forall) state children. This two-level structure implies that SAFA alternate between universal and existential states.

3.5.2. SAFA construction

Now we present a recursive compiler procedure from regex to SAFA using generalized Antimirov derivatives (Section 3.4). Starting with a regex r and alphabet Σ , create an empty SAFA \mathcal{S} with initial state $q_0 \leftarrow r$ by calling the mutually recursive function **SAFA-ADD**(r) as follows.

Function **SAFA-ADD**(\mathcal{S}, r)

1. If state r exists in the SAFA, return. Otherwise add the new state $Q \leftarrow r$.
2. Extract a skip $\mathbf{take}_s(r) = (s, r')$ (Section 3.3) from r .
 - (a) If $s = \{\}$, proceed to step 3.
 - (b) Otherwise, label state r an \exists state ($\lambda_q(r) \leftarrow \exists$) and add e , a new outgoing edge with $\delta \leftarrow (r, e, r')$ and $\lambda_e(e) \leftarrow s$.
 - (c) Then recurse for **SAFA-ADD**(\mathcal{S}, r').
3. If no skip can be extracted, then for each character $\alpha \in \Sigma$ take the generalized Antimirov partial derivative (Figure 3.4) with respect to α to be a Disjunctive Normal Form (DNF) boolean expression $\partial_\alpha(r)$ (Section 3.5.1). In DNF the derivative has two levels, the outer disjunction and inner conjunction ($\partial_\alpha(r) = \bigvee_i (\bigwedge_j r_{i,j})$).
 - (a) For the first level, add i existential \exists states $Q \leftarrow (\bigwedge_j r_{i,j})$ and character transition edges with $\delta \leftarrow (r, e, \bigwedge_j r_{i,j})$ and $\lambda_e(e) \leftarrow \alpha$.
 - (b) For the second level, for each j add a forall \forall state $Q \leftarrow r_{i,j}$ and ϵ edges $\delta \leftarrow ((\bigwedge_j r_{i,j}), \epsilon, r_{i,j})$.
 - (c) Finally recurse for each leaf state **SAFA-ADD**($\mathcal{S}, r_{i,j}$).

Function **SAFA-ADD**(\mathcal{S}, r) looks at the beginning of a regex r and tries to extract the longest possible skip s by calling partial function $\mathbf{take}_s(r) = (s, r')$ (Figure 3.3). If successful, a skip transition is added, otherwise a character transition is added (the two are disjoint). Note, the number of new states added in step 3 is $\mathcal{O}(|\Sigma| \cdot i \cdot j)$. In practice, however, we noticed regex are not nested as much so $i \cdot j$ is small.

3.6. SAFA Satisfaction

Now we can give the semantics of trace satisfaction for SAFA \mathcal{S} , starting with the auxiliary definition of a backtracking solver in Figure 4. A SAFA solver should take a trace σ and check if a SAFA \mathcal{S} recognizes it. In addition, one of our goals was to return efficiently checkable proof objects for trace satisfaction. Proceed to define the type of SAFA proof objects, then define a SAFA solver and finally, combine the SAFA compiler(Figure 3.5.2) with a solver to build an efficient, end-to-end verification pipeline for extended regex with succinct proof objects.

3.6.1. SAFA proof objects

Proof objects should be able to “replay” the SAFA solver without backtracking, and in time sublinear to the length of the trace. If the regular expression is small compared to the length of the trace, then intuitively we should be able to focus on what explicitly appears in the regex and ignore irrelevant parts of the trace.

We define the type of proof objects π as a linked-list of 4-tuples, recording SAFA transitions which are both *successful* and *necessary* to verify trace σ . Every element of π records state **out** transitions with edge **e** to state **in**, and either a matched character or cursor increase value v (the two outcomes are disjoint).

Definition 3.6.1.

$$\pi \in \text{list} \left(\text{out} \in Q, e \in E, \text{in} \in Q, v \in \mathbb{N} \vee \Sigma \right)$$

An efficient checker accepting π simply has to confirm that the SAFA transitions are valid, starting at state q_0 , that the values in π match the values in the trace σ , and the final state is an accepting state.

If $\sigma \in \Sigma^*$ is a string with forward random-access (we can fast-forward from index i to any $i \leq j \leq |\sigma|$) and $i \leq |\sigma|$ a cursor in the string, define the mutually-recursive, parallel function $\text{Solve}(\mathcal{S}, n, i, \sigma)$ in Figure 4. $\text{Solve}(\mathcal{S}, n, i, \sigma)$ is a partial function returning a proof object π iff, at state q , a string σ at index i is accepted by SAFA \mathcal{S} . Otherwise it returns \perp . The solver is defined in two steps, via

Algorithm 2: Efficient checker $\text{Check}(\mathcal{S}, \sigma, \pi)$ for proof object π .

```

Function  $\text{CheckRec}(\mathcal{S}, \sigma, \pi, q, i) :=$ 
  switch  $\pi$  do
    case  $(s, e, t, c \in \Sigma) :: \text{tail}$  do
      | return  $(s, e, t) \in \delta$  and  $\sigma_i = c$  and  $\text{CheckRec}(\mathcal{S}, \sigma, \text{tail}, q, i + 1)$ 
    end
    case  $(s, e, t, n \in \mathbb{N}) :: \text{tail}$  do
      | return  $(s, e, t) \in \delta$  and  $\text{CheckRec}(\mathcal{S}, \sigma, \text{tail}, q, i + n)$ 
    end
    case  $\square$  do
      | return  $q \in \mathcal{F}$  and  $i = |\sigma|$ 
    end
  end
end

Function  $\text{Check}(\mathcal{S}, \sigma, \pi) :=$ 
  if  $(s, e, t, n \in \mathbb{N}) \leftarrow \pi.\text{pop}()$  then
    | return  $s = q_0$  and  $(s, e, t) \in \delta$  and  $\text{CheckRec}(\mathcal{S}, \sigma, \text{tail}, q, n)$ 
  return  $\perp$ 
end

```

two mutually recursive definitions — **SolveEdge** and **Solve**.

Function **Solve** calls **SolveEdge** in Figure 3 for every edge in the SAFA, and **SolveEdge** calls **Solve** for every destination state.

The solver leverages quantifiers and skips in SAFA to get a high-performance parallel implementation. **Solve** handles the two types of states in \mathcal{S} calling **SolveEdge** in parallel. For universal states ($\lambda_q(n) = \forall$) one child thread is spawned for each transition $(s, e, t) \in \text{outgoing}(n)$. Then if every thread reaches an accepting state, the solutions are combined into one π . Due to the universal quantifier, all solutions must reach an accepting state and all solutions must appear in π and checked individually. For existential states ($\lambda_q(n) = \exists$) one child thread is spawned for each transition $(s, e, t) \in \text{outgoing}(n)$, however, only one thread is required to reach a solution **sol**. The first thread to find a solution returns the solution, killing all other threads which may or may not succeed.

The same is true for **SolveEdge**, an auxiliary recursive function that handles the two types of edges in \mathcal{S} . **SolveEdge** handles character set \mathcal{C} edges in a straightforward way, by checking $\sigma_i \in \mathcal{C}$. For

Algorithm 3: SAFA solver (auxiliary), recursively processes edges.

```

Function SolveEdge( $\mathcal{S}, e, \text{from}, \text{to}, i, \sigma$ ) :=
  switch  $\lambda_e(e)$  do
    case CharSet( $C$ ) do
      if  $\sigma_i \in C$  then
        // recurse on next index
        if  $\text{tail} \leftarrow \text{Solve}(\text{to}, i + 1, \text{doc})$  then
          | return ( $\text{from}, e, \text{to}, D_i$ ) ::  $\text{tail}$ ;          // Add character  $\sigma_i$  to solution
        return  $\perp$ ;
      end
    case Skip( $\text{range}$ ) do
       $\text{candidates} \leftarrow \{i + n \mid n \in \text{range} \text{ and } i + n \leq |\sigma|\}$ ;
      // Race threads to find a matching  $n$ 
      parallel for  $n \in \text{candidates}$  do spawn
        | if  $\text{tail} \leftarrow \text{Solve}(\text{to}, i + n, \sigma)$  then
          | | return ( $\text{from}, e, \text{to}, n$ ) ::  $\text{tail}$ ;          // Add skip value  $n$  to solution
        end
      end
    end
  end
end

```

skip a edge s , **SolveEdge** tries different skip values $n \in s$ in parallel. The first thread for which n leads to an accepting solution is returned, killing other threads who may lead to a solution or may not.

Note, common regular expression matching programs like **grep** have a different, deterministic behavior. When **grep** is faced with two choices of matches that work, it always picks the longest atomic match (greedy). For example if $r = "(a \vee aa) * "$ and $\sigma = "aaaa"$ it will pick $aa \rightarrow aa$ instead of $a \rightarrow a \rightarrow a \rightarrow a$, because it leads to longer atomic submatches aa instead of a . The SAFA solver Algorithm 4 will nondeterministically pick any of the accepted solutions, for our example it might return $aa \rightarrow a \rightarrow a$ or $a \rightarrow aa \rightarrow a$ in the form of π . Determinism is an implementation detail not covered by the definition of regular languages — indeed both solutions above can prove $\sigma \in \mathcal{L}[[r]]$.

Finally, with the definitions of proof producing **Solve**($\mathcal{S}, n, i, \sigma$) in Figure 4 and incorporating the SAFA compiler **SAFA-ADD** we can finally define an efficient, proof producing, satisfaction relation

Algorithm 4: SAFA solver (top-level), recursively process states.

```

Function Solve( $\mathcal{S}, n, i, \sigma$ ) :=
  if  $q \in \mathcal{F}$  and  $i = |\sigma|$  then
    | return  $\square$ ;                                // Termination in accepting state, success
  end
  if outgoing( $n$ ) =  $\emptyset$  then
    | return  $\perp$ ;                                //  $n$  is a sink state, fail
  end
  switch  $\lambda_q(n)$  do
    case  $\forall$  do
      | subsolutions  $\leftarrow \emptyset$ ;
      | parallel for  $(s, e, t) \in \text{outgoing}(n)$  do spawn
      |   | if sol  $\leftarrow \text{SolveEdge}(\mathcal{S}, e, s, t, i, \sigma)$  then
      |   |   | subsolutions  $\leftarrow \text{subolutions} \cup \{\text{sol}\}$ ;
      |   |   end
      |   end
      |   if |subolutions| = |outgoing( $n$ )| then
      |   |   | return flatten(subolutions)
      |   |   return  $\perp$ ;
      | end
    case  $\exists$  do
      | parallel for  $(s, e, t) \in \text{outgoing}(n)$  do spawn
      |   | if sol  $\leftarrow \text{SolveEdge}(\mathcal{S}, e, s, t, i, \sigma)$  then
      |   |   | return sol;                        // Only one thread returns, the rest are killed
      |   |   end
      |   end
      |   return  $\perp$ ;
    end
  end
end

```

for extended regular expression r and finite traces σ . Define the satisfaction relation **Satisfy**, by combining the compiler and solver, in Algorithm 5.

3.7. Conclusion and Future Work

We have introduced Skipping Alternating Finite Automata (SAFA) (Section 3.2), a novel extension to alternating finite automata that enables efficient verification of regular specifications with succinct proof objects. SAFA builds on the expressive power of AFA by adding skip transitions - a mechanism to handle wildcard regions in regular expressions without requiring explicit character matching. This

addition allows SAFA to represent extended regular expressions compactly while maintaining their ability to generate small, easily checkable proof certificates.

The key innovations of SAFA are threefold. First, our use of generalized Antimirov derivatives (Section 3.5.1) combined with alternation between universal and existential states provides a natural way to handle boolean operations while avoiding the state explosion that typically occurs due to distributivity of conjunction, disjunction, and concatenation. Second, skip transitions (Section 3.3) provide an elegant mechanism to handle wildcard patterns without checking individual characters, making verification more efficient for traces with large wildcard regions. Third, our multithreaded solver (Algorithm 4) leverages both SAFA state quantifiers and the independence of skip transitions to enable efficient multithreaded matching, where different potential solutions can be explored simultaneously.

We have demonstrated a complete verification pipeline for regular specifications using SAFA (Algorithm 5) including:

1. A compiler from extended regular expressions to SAFA based on generalized Antimirov partial derivatives (Section 3.5).
2. A parallel solver that leverages both skip transitions and alternation to efficiently confirm traces match (Algorithm 4).
3. A proof checker that can verify succinct certificates in sublinear time, faster than rerunning the trace solver (Algorithm 5).

This combination of features makes SAFA particularly well-suited for applications with regular

Algorithm 5: Extended regular expression satisfaction with proof objects using SAFA

Input: Regular expression r , alphabet Σ , trace σ

Output: Proof object π or \perp

Function Satisfy(r, Σ, σ):=

```

    // Initialize empty SAFA
     $S \leftarrow (Q : \emptyset, E : \emptyset, \Sigma, \lambda_q : \emptyset, \lambda_E : \emptyset, \delta : \emptyset, q_0 : r, \mathcal{F} : \emptyset);$ 
    SAFA-ADD( $S, r$ );                                     // Compile  $r$  into SAFA  $S$ 
    return Solve( $S, q_0, 0, \sigma$ );                       // Match  $S$  on  $\sigma$  starting on index 0
end
```

expressions where efficient verification with proof objects is required. Indeed, SAFA has been successfully applied in the Reef compiler [AIM⁺24], which generates zero-knowledge cryptographic certificates for regex matching over secret documents. In this application, SAFA’s ability to generate compact certificates that can be checked independently translates directly to smaller proof sizes and better performance in real-world systems. From network middleboxes that verify traffic patterns without seeing packet contents, to secure DNA matching systems that check gene patterns while keeping sequences private, to password checkers that verify complexity rules without revealing the password itself [AIM⁺24].

Looking ahead, SAFA opens up interesting directions for future work, particularly in extending these techniques to other classes of automata used in model checking. One significant limitation of model checking compared to deductive verification has been the lack of proof certificates - a gap that SAFA’s certificate generation approach could help bridge. Applying skip transitions and alternation to Büchi automata [Var07] could enable more efficient verification of infinite-state systems with succinct proofs. Finally, the fundamental ideas of skipping and alternation [CKS81] transcend regular languages, with numerous applications in more expressive models of computation — from context-free grammars to Turing machines. We believe skipping and alternation can potentially bring efficient verification with compact proofs to these broader domains.

Definition 3.7.1.

$$\begin{aligned}
[a_1, b_1] \cup [a_2, b_2] &= \begin{cases} \{[\min(a_1, a_2), \max(b_1, b_2)]\} & \max(a_1, a_2) \leq \min(b_1, b_2) \\ \{[a_2, b_2], [a_1, b_1]\} & b_2 < a_1 \\ \{[a_1, b_1], [a_2, a_2]\} & \text{otherwise} \end{cases} \\
[a_1, \infty) \cup [a_2, b_2] &= \begin{cases} \{[a_2, b_2], [a_1, \infty)\} & b_2 < a_1 \\ \{[a_1, \infty)\} & b_2 \geq a_1 \leq a_2 \\ \{[a_2, \infty)\} & \text{otherwise} \end{cases} \\
[a_1, b_1] \cup [a_2, \infty) &= \begin{cases} \{[a_1, b_1], [a_2, \infty)\} & b_1 < a_2 \\ \{[a_2, \infty)\} & b_1 \geq a_2 \leq a_1 \\ \{[a_1, \infty)\} & \text{otherwise} \end{cases} \\
[a_1, \infty) \cup [a_2, \infty) &= \begin{cases} \{[a_1, \infty)\} & a_1 \leq a_2 \\ \{[a_2, \infty)\} & \text{otherwise} \end{cases} \\
\{\} \cup s &= s \\
\{i, \overline{i_n}\} \cup s &= \{i \cup i_s \mid i_s \in s\} \cup \{\overline{i_n}\} \\
[a_1, \infty) \cdot [a_2, \infty) &= [a_1 + a_2, \infty) \\
[a_1, b_1] \cdot [a_2, \infty) &= [a_1 + a_2, \infty) \\
[a_1, \infty) \cdot [a_2, b_2] &= [a_1 + a_2, \infty) \\
[a_1, b_1] \cdot [a_2, b_2] &= [a_1 + a_2, b_1 + b_2] \\
s_1 \cdot s_2 &= \bigcup \{i_1 \cdot i_2 \mid i_1 \in s_1, i_2 \in s_2\} \\
\{\}^* &= \{\} \\
\epsilon^* &= \epsilon \\
s^* &= \begin{cases} \{[0, \infty)\} & \text{if } 1 \in s \\ \{\} & \text{otherwise} \end{cases} \\
\{\}\{n, m\} &= \{\} \\
\{\overline{i_v}\}\{n, m\} &= \begin{cases} \epsilon & \text{if } n = m = 0 \\ \{\overline{i_v}\} \cdot \{\overline{i_v}\}\{0, m-1\} & \text{else if } n = 0 \\ \{\overline{i_v}\} \cdot \{\overline{i_v}\}\{n-1, m-1\} & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.5: Interval sets are closed under *union* ($s \cup s'$), *concatenation* ($s \cdot s'$), *Kleene-star* (s^*) and *bounded repetition* ($s\{n, m\}$).

CHAPTER 4

Structural Program Verification with Temporal Logic (Ticl)

4.1. Safety and Liveness

Mechanized program verification can be used to formally guarantee that executable code satisfies important properties, most notably *liveness* and *safety* properties. Liveness properties (“a good thing happens”) include *termination* and *fairness*, as well as *always-eventually* properties, and 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 *always-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 [AM01, JKJ⁺18, AHM⁺17, MAA⁺19, WAH⁺22, SZ21].

Arguably, the widespread success of mechanized safety verification has been due to the development of structural program logics, such as Hoare logic. The basic construct, the Hoare triple $\{P\} c \{Q\}$, specifies that if the precondition P holds before executing the command c , then the postcondition Q will hold afterward. Hoare logic has three crucial benefits that significantly simplifies the process of proving safety: (1) *modularity*; (2) *composition*; and (3) *structural proof rules*. Modularity allows one to perform local reasoning by breaking down complex programs into small modular components, making it easier to verify the correctness of individual parts without needing to understand the whole. Composition is given by the sequence rule, which combines triples $\{P\} c_1 \{Q\}$ and $\{Q\} c_2 \{R\}$ to get $\{P\} c_1; c_2 \{R\}$, building bigger proofs from smaller subproofs. Structural Hoare rules like assignment ($x := a$), conditionals (**if** c **then** a **else** b), and loops (**while** c **do** b) allow reasoning over standard program constructs while hiding their semantic interpretations.

Unfortunately, the 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* [Pnu77, EC82, BCG88, AHK02, Lam94, KP84] these tend to be primarily

focused on *semantic models* of program execution, for example coinductive traces and transition systems [Pnu77, EC82, SVW87, DNV90, AHK02, DS16, HHK⁺15]. Reasoning about coinductive traces and transition systems in a proof assistant is arduous, requiring nested induction and coinduction techniques and deep understanding of complex mathematical concepts like the Knaster-Tarski lemma and the proof assistant’s mechanics (e.g. the guardedness checker). Furthermore, the benefits of *modularity*, *composition* and *structural proof rules* of Hoare Logic do not apply in the semantic domain. Certain liveness properties have been studied in a *syntactic* setting [LCK⁺23, DSFG21, LF16] but these are limited in expressivity. There has never been a general approach to mechanized, structural, temporal logic proofs.

Contributions: We introduce *Temporal Interaction and Choice Logic* (**ticl**), a new structural program logic inspired by *computation tree logic* (CTL) [EC82]. **Ticl** allows proving rich temporal properties compositionally, using syntax-driven lemmas, while hiding much of the complexity associated with (co-)inductive proof techniques behind high-level, reusable, structural proof lemmas. Our **ticl** framework packages over 15K lines of nested (co-)inductive proofs and definitions—in around 50 high-level lemmas that are easy to use. We posit this metatheory is rich enough to formally prove useful safety and liveness specifications and demonstrate its use with examples from sequential, concurrent and distributed programming: imperative programs with queues, secure shared memory, and a simple distributed consensus protocol.

Our programming languages are based on a denotational model in the **ITree** family [XZH⁺19, CHH⁺23] capable of expressing infinite, non-deterministic, effectful programs (Figure 4.3), allowing for expressive programs with loops, concurrency, mutable state and message passing. Our development is formalized in the Coq proof assistant [The24], relying only on the `eq_rect_eq` axiom, also known as *uniqueness of identity proofs*. **ticl** is released under an open-source license ¹.

Related Work: Step-indexed logical relation frameworks like Iris [AM01, JKJ⁺18], can prove safety but not liveness properties. More recently, transfinite extensions to step-indexing [SGG⁺21] made it possible to prove *always* properties but not *always-eventually* properties. Fair Operational

¹<https://github.com/vellvm/ticl>

Semantics [LCK⁺23] are limited to binary *always-eventually* properties, specifically *good* vs *bad* events and do not generalize to arbitrary liveness properties. Other works on fairness including TaDa-Live [DSFG21] and LiLi [LF16], have limited expressive power and do not provide a general framework for arbitrary temporal specifications. Some deductive verification frameworks for temporal properties, for example Cyclist [TB17] lack support for expressing terminating programs. Temporal Rewriting Logic (TLR) [Mes08] and the Maude language [Mes92] also do not support finite or deadlocked programs and operate on the level of models, not on the level of executable programs.

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 `Ltac` tactics and inversion lemmas we developed. Still as we report in the feature table of Figure 4.14, completeness of `ticl` is an open question we leave for future work.

4.2. Low-level temporal proofs

Definition <code>rotate</code> := <code>do (</code> <code> $x \leftarrow \text{pop}$; $\text{push } x$</code> <code>) while (true)</code>	Theorem <code>rotate_agaf</code> : $\forall(x: T) (q: \text{list } T),$ $\langle (\text{instrQ rotate } (q ++ [x])), \text{Pure} \mid= \text{AG AF obs } (\lambda \text{hd} \Rightarrow \text{hd} = x) \rangle.$
--	--

Figure 4.1: Program `rotate` runs forever, pops an element from the head and appends it. The specification (`rotate_agaf`) is *always-eventually* x will appear in the head position.

We now illustrate the challenges of proving a simple liveness property for a small program in Coq. Consider a task server that maintains a queue of re-entrant tasks and processes them in order, also called a round-robbin queue. The `rotate` program in Figure 4.1 implements round-robbin —a simple infinite loop removes a task from the head of the queue and inserts it at the end. Our goal is to prove a queue task x will *always-eventually* appear in the head position (**AG AF** using CTL notation [EC82]). A common technique for working with infinite programs is denoting to a coinductive tree of events. For `rotate`, the tree degenerates to a coinductive stream of alternating `[pop, push, pop...]` events. The target specification (“always-eventually x appears in the head position”) is expressed in terms

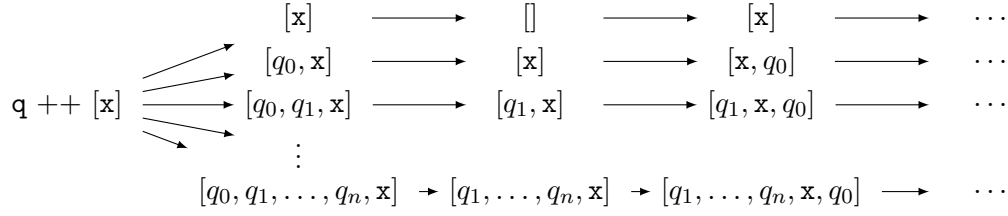


Figure 4.2: Instrumentation of `rotate` with initial state $q ++ [x]$.

of queues (states), not events, so semantic interpretation of events to states given initial state ($q ++ [x]$) is necessary. Glossing over the semantics of queues, the translation of queue events to states matches the degenerate tree structure of Figure 4.2, where each infinite trace depends on the length of the initial q .

We must then express the property “always-eventually x appears in the head position” as a nested inductive/coinductive predicate over the degenerate tree structure in Figure 4.2, then prove it by nested induction (on the length of q) and coinduction on each trace. The proof is not easy—working directly with trees of traces and low-level induction/coinduction tactics we lose the benefits of *modular* and *structural* proof rules from program logics. Even the trivial-looking example `rotate` requires a non-trivial amount of infrastructure to prove, most of which is not reusable for other programs and specifications.

In contrast, with `ticl`, proving the `rotate` example from Figure 4.1 is reduced to a straightforward application of the *invariance* rule for infinite do-while loops that we have already proved (for a preview, see Figure 4.24).

4.3. Definitions

Our goal is to build an expressive, temporal logic in the style of CTL [EC82] using a coinductive tree structure in place of the model. If we succeed, then we will be able to write and prove temporal properties over effectful, nondeterministic, possibly non-terminating programs.

We first give a definition of the coinductive model `ictree` and its algebraic theory. We define *instrumentation* over `ictree` structures as a way of evaluating programs while maintaining trace

$$\begin{aligned}
\mathbf{ictree} &\in (\text{Type} \rightarrow \text{Type}) \rightarrow \text{Type} \rightarrow \text{Type} \\
\mathbf{ictree}_{E, X} &\stackrel{\text{coind}}{=} \mid \mathbf{Ret} (x \in X) \quad \mid \mathbf{Vis} (X \in \text{Type}) (e \in E X) (k \in X \rightarrow \mathbf{ictree}_{E, X}) \\
&\quad \mid \mathbf{Guard} (t \in \mathbf{ictree}_{E, X}) \quad \mid \mathbf{Br} (n \in \mathbb{N}) (k \in \text{fin}' n \rightarrow \mathbf{ictree}_{E, X}) \\
\text{fin}' (n \in \mathbb{N}) &\in \text{Type} = \text{fin} (S n) \\
\emptyset &\in \mathbf{ictree}_{E, X} \stackrel{\text{coind}}{=} \mathbf{Guard} \emptyset \\
\gg &\in \mathbf{ictree}_{E, X} \rightarrow (X \rightarrow \mathbf{ictree}_{E, Y}) \rightarrow \mathbf{ictree}_{E, Y} \\
(\mathbf{Ret} x) \gg f &= f x, \quad (\mathbf{Vis} X e k) \gg f \stackrel{\text{coind}}{=} \mathbf{Vis} X e (\lambda (x \in X) \Rightarrow (k x) \gg f) \\
(\mathbf{Guard} t) \gg f &\stackrel{\text{coind}}{=} \mathbf{Guard} (t \gg f), \quad (\mathbf{Br} n k) \gg f \stackrel{\text{coind}}{=} \mathbf{Br} n (\lambda (i \in \text{fin}' n) \Rightarrow (k i) \gg f) \\
\mathbf{iter} &\in (I \rightarrow \mathbf{ictree}_{E, I+R}) \rightarrow I \rightarrow \mathbf{ictree}_{E, R} \\
\mathbf{iter} \text{ step } i &\stackrel{\text{coind}}{=} (\text{step } i) \gg \lambda (lr \in I + R) \Rightarrow \begin{cases} \mathbf{Guard} (\mathbf{iter} \text{ step } i'), & lr = \text{inl } i' \\ \mathbf{Ret} (r), & lr = \text{inr } r \end{cases} \\
\mathbf{trigger} (e \in E X) &\in \mathbf{ictree}_{E, X} = \mathbf{Vis} X e (\lambda (x \in X) \Rightarrow \mathbf{Ret} x) \\
\mathbf{branch} (n \in \mathbb{N}) &\in \mathbf{ictree}_{E, \text{fin}' n} = \mathbf{Br} n (\lambda (i \in \text{fin}' n) \Rightarrow \mathbf{Ret} i) \\
\oplus &\in \mathbf{ictree}_{E, X} \rightarrow \mathbf{ictree}_{E, X} \rightarrow \mathbf{ictree}_{E, X} \\
1 \oplus r &= \mathbf{Br} _ \left(\lambda (i \in \text{fin } 2) \Rightarrow \begin{cases} 1, & i = F_1 \\ r, & i = FS F_1 \end{cases} \right)
\end{aligned}$$

Figure 4.3: Definitions and core combinators for **ictree**.

information. We give a Kripke small-step semantics to **ictree** and use the stepping relation to define the syntax and semantics of **ticl** formulas. With regards to syntax, we introduce two syntactic categories of formulas; *prefix* formulas that capture the prefix of a tree (or infinite trees, for example *always*) and *suffix* formulas that capture terminating trees.

4.3.1. The **ictree** denotational model

Core definitions and up-to-guard equivalence

Interaction Trees and Choice Trees [XZH⁺19, CHH⁺23] are commonly used to reason about non-terminating, nondeterministic, interactive programs. We define the **ictree** structure inspired by Choice Trees [CHH⁺23] in Figure 4.3. The coinductive **ictree** structure has visible event nodes (**Vis**), silent τ nodes (**Guard**) and finite non-deterministic choice with positive arity (**Br**). Finite non-determinism with positive arity is more limited compared to the dual notion of non-determinism in Choice Trees, but sufficient to verify our use cases.

Guard nodes much like τ nodes for ITrees are silent. The *stuck* (\emptyset) **ictree** represents the deadlocked state that cannot make any progress. Following the same methodology as ITrees [XZH⁺19] define

a coinductive *up-to-guard* equivalence relation (\sim) that ignores a finite number of guards. The structure is a monad ($\gg=$, **Ret**) and the **iter** combinator encodes both finite and infinite loops. We prove the monad equations hold with regards to \sim , among others in Figure 4.4. Equational reasoning on **ictree** structures is a powerful proof technique used extensively in our development and in combination with temporal reasoning in the examples later.

$$\begin{array}{c}
\frac{}{t \sim t} \text{SBREFL} \qquad \frac{t \sim u}{u \sim t} \text{BSYM} \qquad \frac{t \sim u \quad u \sim v}{t \sim v} \text{SBTRANS} \qquad \frac{}{\text{Guard } t \sim t} \text{SBGUARD} \\
\\
\frac{t \sim u \quad (\forall x, g \ x \sim k \ x)}{t \gg= g \sim u \gg= k} \text{SBIND} \qquad \frac{}{\text{Ret } v \gg= k \sim k \ v} \text{SBINDL} \qquad \frac{}{x \leftarrow \mathbf{t};; \text{Ret } x \sim t} \text{SBINDR} \\
\\
\frac{}{(t \gg= k) \gg= l \sim t \gg= (\lambda x \Rightarrow k \ x \gg= l)} \text{SBINDASSOC} \qquad \frac{x = y}{\text{Ret } x \sim \text{Ret } y} \text{SBRET} \\
\\
\frac{\forall x, h \ x \sim k \ x}{\text{Vis } e \ h \sim \text{Vis } e \ k} \text{SBVIS} \qquad \frac{\forall x, h \ x \sim k \ x}{\text{Br } n \ h \sim \text{Br } n \ k} \text{SBBR}
\end{array}$$

Figure 4.4: Equational theory for **ictree** with respect to *up-to-guard* equivalence relation.

Semantic interpretation and instrumentation

Vis events are uninterpreted events. To reason about their meaning a semantic handler $\mathbf{h}: \mathbf{E} \rightsquigarrow \mathbf{M}$ must be provided during interpretation, where \mathbf{M} is a monad compatible with **ictree** structures. In this work we introduce *instrumentation*, a special case of semantic interpretation where every event ($\mathbf{e}: \mathbf{E} \ \mathbf{X}$) is interpreted over a state monad transformer (**stateT** \mathbf{S}), and leaves behind a trace of *observation* events (\log_W) in Figure 4.5. We call the monad $\mathbf{InstrM}_{\mathbf{S}, \mathbf{W}}$ the *instrumentation monad*. Instrumentation events \log_W return the **unit** type and cannot be interpreted further, we can simply erase them without changing the semantics of program evaluation. The intuition for instrumentation is in Figure 4.2, **instr** is precisely the transformation from events to states using the semantics of queue operations and an initial state.

Note in $\mathbf{InstrM}_{\mathbf{S}, \mathbf{W}}$ the type of concrete state (\mathbf{S}) is different from the type of observations (\mathbf{W}). In program verification a proof often needs to track auxiliary state for the sake of maintaining a strong invariant and proving a goal, called *ghost state*. For example, to prove liveness of the distributed

```

logW ∈ Type → Type = | Log (w ∈ W) ∈ logW unit
InstrMS,W ∈ Type → Type = stateT S ictreelogW
instr ∈ (E ∼ InstrMS,W) → ictreeE ∼ InstrMS,W
  instr h (Ret x) s = Ret (x, s),   instr h (Guard t) s  $\stackrel{\text{coind}}{=}$  Guard (instr h t s)
  instr h (Vis X e k) s  $\stackrel{\text{coind}}{=}$  (h e s) >>= (λ '(x ∈ X, s' ∈ S) ⇒ Guard (instr h (k x) s'))
  instr h (Br n k) s  $\stackrel{\text{coind}}{=}$  Br n (λ (i ∈ fin' n) ⇒ instr h (k i) s)

```

Figure 4.5: Instrumentation of an $\text{ictree}_{E, X}$ with \log_W events over state S produces an instrumentation monad $\text{InstrM}_{S,W}$.

consensus protocol in Section 4.5.3, we must keep track of each delivered message using *ghost state*, then show the sequence of delivered messages is monotonically decreasing with respect to some metric, until consensus is reached.

4.3.2. Kripke small-step semantics

Temporal logics are usually defined over infinite traces, finite [DGV⁺13] traces or various transition systems. We define a Kripke transition system as the base for ticl . Before defining the transition relation we must first define the notion of a *world* (\mathcal{W}_E). A world is an enumeration with a partial order (Figure 4.6) that “remembers” events of type **E** and the “status” of the transition system. A **Pure** world indicates no events observed yet, a world (**Obs** $e v$) remembers the last observed event (**e**: **E** **X**) and response (**v**: **X**), a world **Val** x indicates the return value (**x**) of a pure program, and world **Finish** $e v x$ indicates the return value (**x**) of an effectful computation with last event (**e**: **E** **X**) and response (**v**: **X**). Worlds are either **done** (**Val**, **Finish**) or **not_done** (**Pure**, **Obs**).

Then, the Kripke transition relation is an irreflexive, binary, inductive relation, over pairs of **ictree** and worlds ($\mapsto \in \text{relation}(\text{ictree}_{E, X} * \mathcal{W}_E)$) defined in Figure 4.7. Transitions are inductively defined over finite **Guard** nodes and \emptyset trees cannot transition. Only **not_done** worlds can transition. If a **Pure** world transitions to an observation **Obs** $e v$ then it can never transition to a **Pure** world again. The restrictions on worlds induce the partial order in Figure 4.6.

Transitioning to a **done** world and \emptyset means the program terminated and cannot transition any further. This is a departure from the left-total (i.e: $\forall m, \exists m', R m m'$) Kripke transition relations that most often appear in literature [DNV90, EH86]. The reason for this departure is the semantics

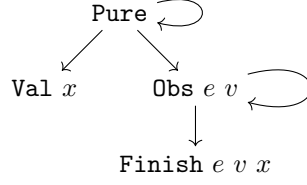


Figure 4.6: Kripke world \mathcal{W}_E parametrized by event type \mathbf{E} .

of *bind* (\gg). From the first monad law ($\mathbf{Ret} \ v \gg k \sim k \ v$) in Figure 4.4, if $\mathbf{Ret} \ v \gg k$ was to take a number of steps and the transition relation was left-total, then $\mathbf{Ret} \ v$ would step forever, never returning the value to the continuation (k) so it could step as well. This behavior does not agree with the stepping semantics of the equivalent tree ($k \ v$), which is why totality of Kripke transitions does not work for monadic structures.

Our non-total Kripke semantics are a simple variation on finite trace LTL [DGV⁺13, AMO19] which is well-studied. However, we are not aware that the connection from finite traces to monadic composition has been made before. We show how monadic composition interacts with our Kripke semantics in Figure 4.8.

The lemma EXEQUIV in Figure 4.8 is of particular interest. It shows transitions are not \sim -invariant, but we can always provide an \sim -equivalent $\mathbf{ictree}_{E, X}$ to get an equivalent the transition. We recover \sim -invariance at the level of \mathbf{ticl} entailment in subsection 4.3.3, allowing us to reason modulo and any finite number of guards.

$$\begin{array}{c}
\mathcal{W}_E \in \text{Type} = \mathbf{Pure} \mid \mathbf{Obs} \ e \ v \mid \mathbf{Val} \ x \mid \mathbf{Finish} \ e \ v \ x \\
\\
\begin{array}{c}
\text{not_done } \mathbf{Pure} \qquad \text{not_done } (\mathbf{Obs} \ e \ v) \qquad \frac{[t, w] \mapsto [t', w']}{[\mathbf{Guard} \ t, w] \mapsto [t', w']} \qquad \frac{\text{not_done } w \quad 0 \leq i < n}{[\mathbf{Br} \ n \ k, w] \mapsto [k \ i, w]} \\
\\
\frac{\text{not_done } w}{[\mathbf{Vis} \ e \ k, w] \mapsto [k \ v, \mathbf{Obs} \ e \ v]} \qquad [\mathbf{Ret} \ x, \mathbf{Pure}] \mapsto [\emptyset, \mathbf{Val} \ x] \qquad [\mathbf{Ret} \ x, \mathbf{Obs} \ e \ v] \mapsto [\emptyset, \mathbf{Finish} \ e \ v \ x]
\end{array}
\end{array}$$

Figure 4.7: Kripke semantics of ctrees and **not_done** world predicate.

4.3.3. Ticl syntax & semantics

Syntax

Equipped with our Kripke semantics, we next define the syntax of **ticl** formulas in Figure 4.9. We first we give an informal definition of each operator. We will use the metavariables p, q to refer to either prefix or a suffix formulas from this point forward.

- **now** ($P \in \mathcal{W}_E \rightarrow \mathbb{P}$) : Base case, the current world w is **not_done** and predicate P holds.
- **done** ($P_X \in X \rightarrow \mathcal{W}_E \rightarrow \mathbb{P}$) : Base case, the current world w is **done** and P_X holds.
- φ **AN** q : Formula φ holds, then next q holds for all worlds accessible in one step.
- φ **EN** q : Formula φ holds, then next q holds for at least one world accessible in one step.
- φ **AU** q : Formula φ holds for all paths, until q *eventually* holds, or q holds right now.
- φ **EU** q : Formula φ holds for at least one path, until q *eventually* holds, or q holds right now.
- **AG** φ : Formula φ always holds in all paths and all paths are infinite.
- **EG** φ : There exists at least one infinite path for which Formula φ always holds.
- $p \wedge q$: Both p and q hold.
- $p \vee q$: Either p or q hold.

There are two syntactic classes in **ticl**, *prefix* formulas φ and *suffix* formulas ψ_X . Prefix formulas represent temporal properties satisfiable by an **ictree** prefix, meaning they must be satisfied before the **ictree** returns. On the other hand, suffix formulas are satisfiable only by a terminating **ictree** and need to observe its return value and world w that is **done** to be satisfied. Suffix formulas are a syntactic superclass of prefix formulas as the binary operators φ **AN** ψ_X, φ **AU** ψ_X, φ **EU** ψ_X

$$\begin{array}{c}
\frac{s \sim t \quad [s, w] \mapsto [s', w']}{\exists t', [t, w] \mapsto [t', w'] \wedge s' \sim t'} \text{EXEQUIV} \qquad \frac{[t, w] \mapsto [t', w'] \quad \text{not_done } w'}{[x \leftarrow \mathbf{t};; k \ x, w] \mapsto [x \leftarrow \mathbf{t}';; k \ x, w']} \\
\\
\frac{[t, w] \mapsto [\emptyset, \mathbf{val} \ x] \quad [k \ x, w] \mapsto [t', w']}{[x \leftarrow \mathbf{t};; k \ x, w] \mapsto [t', w']} \qquad \frac{[t, w] \mapsto [\emptyset, \mathbf{Finish} \ e \ v \ x] \quad [k \ x, w] \mapsto [t', w']}{[x \leftarrow \mathbf{t};; k \ x, w] \mapsto [t', w']}
\end{array}$$

Figure 4.8: Derived lemmas for kripke transitions for **ictree**.

$\varphi, \varphi' ::=$	$\psi_x, \psi'_x ::=$	
$\text{now } (P \in \mathcal{W}_E \rightarrow \mathbb{P})$	$\text{done } (P_x \in X \rightarrow \mathcal{W}_E \rightarrow \mathbb{P})$	$\top = \text{now } (\lambda _ . \top)$
$\varphi \text{ AN } \varphi'$	$\varphi \text{ AN } \psi_x$	$\perp = \text{now } (\lambda _ . \perp)$
$\varphi \text{ EN } \varphi'$	$\varphi \text{ EN } \psi_x$	$\top' = \text{done } (\lambda _ _ . \top)$
$\varphi \text{ AU } \varphi'$	$\varphi \text{ AU } \psi_x$	$\perp' = \text{done } (\lambda _ _ . \perp)$
$\varphi \text{ EU } \varphi'$	$\varphi \text{ EU } \psi_x$	$\text{AX } p = \top \text{ AN } p$
$\text{AG } \varphi$	$\psi_x \wedge \psi'_x$	$\text{EX } p = \top \text{ EN } p$
$\text{EG } \varphi$	$\psi_x \vee \psi'_x$	$\text{AF } p = \top \text{ AU } p$
$\varphi \wedge \varphi'$		$\text{EF } p = \top \text{ EU } p$
$\varphi \vee \varphi'$		
$\text{pure} = \text{now } (\lambda w. w = \text{Pure})$	$\text{val } p = \text{done } (\lambda x w. w = \text{Val } x \wedge p \ x)$	
$\text{obs } p = \text{now } (\lambda w. w = \text{Obs } e \ v \ \wedge \ p \ e \ v)$	$\text{finish } p = \text{done } (\lambda x w. w = \text{Finish } e \ v \ x \ \wedge \ p \ x \ e \ v)$	
	$\text{done}_= \ x \ w = \text{done } (\lambda x' w'. w = w' \wedge x = x')$	

Figure 4.9: Syntax of **ticl** prefix formulas (φ), suffix formulas (ψ_x) and useful syntactic notations.

include prefix formulas on their left-hand argument. Due to their appearance to the left-side of temporal operators we also refer to prefix formulas φ as *left* formulas with and to suffix formulas ψ_x as *right* formulas. Since suffix formulas (ψ_x) capture return values they are parametrized by the return value of an **ictree** _{E, X} .

The reasoning behind the use of dual syntax is motivated by the syntax-driven **bind** and **iter** lemmas in Section 4.4.1, as there are different proof obligations for formulas that are satisfiable by finite and infinite trees. A difference between **ticl** syntax and CTL syntax is the *next* operators **AN**, **EN** operators are binary, instead of the the unary **AX**, **EX** operators of CTL. We reclaim their unary counterparts with syntactic notations at the bottom of Figure 4.9.

Semantics of entailment

Assign semantic meaning to **ticl** formulas with two ternary entailment relations \models_L, \models_R defined inductively on the structure of formulas in Figure 4.9. The goal is to build nested inductive and coinductive predicates of type **ictree** _{E, X} $\rightarrow \mathcal{W}_E \rightarrow \mathbb{P}$. To make clear the distinction between induction on **ticl** formulas and path induction for the *until* operators **AU**, **EU**, we first define the *shallow* predicates in the proof assistant’s metalanguage in Figure 4.10.

Definitions **anc**, **enc**, **auc**, **euc** are *higher-order predicates*, they take two predicates of type **ictree** _{E, X} $\rightarrow \mathcal{W}_E \rightarrow \mathbb{P}$ as arguments and transport them under their modal operator to get a “future” predicate

$$\begin{aligned}
\mathbf{can_step} \, t \, w \in \mathbb{P} &= \exists t', w', [t, w] \mapsto [t', w'] \\
\mathbf{anc} \, P \, Q \, t \, w \in \mathbb{P} &= P \, t \, w \wedge \mathbf{can_step} \, t \, w \wedge \forall t', w', [t, w] \mapsto [t', w'] \rightarrow Q \, t' \, w' \\
\mathbf{enc} \, P \, Q \, t \, w \in \mathbb{P} &= P \, t \, w \wedge \exists t', w', [t, w] \mapsto [t', w'] \wedge Q \, t' \, w'
\end{aligned}$$

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

Figure 4.10: *Next* (**anc**, **enc**) and inductive *Until* (**auc**, **euc**) shallow predicates used to define \models_L, \models_R .

of the same type. The restriction **can_step** on *all-next* (**anc**) asserts the existence of at least one transition, we call this *strong-next*, and is necessary because our transition relation is not left-total and allows for deadlocked states (\emptyset). If we omit **can_step** $t \, w$ then $\langle \emptyset, w \models_L \mathbf{AX} \perp \rangle$ can be trivially proven; by introducing the (contradictory) hypothesis $[\emptyset, w] \mapsto [t', w']$ then $\langle t', w' \models_L \perp \rangle$ is provable. Since every terminating program will eventually step to \emptyset , eventually false would be always provable. However, with *strong-next* this is solved and **ticl** is sound even in the face of deadlocked states.

Finally, define entailment by induction on the structure of formulas $\varphi, \psi_{\mathbf{x}}$ in Figure 4.11. Note the inner induction for the *until* operators **AU**, **EU** by calling their shallow counterparts, and inner coinduction for the *always* using the *greatest fixpoint* **gfp** operator (we use the coinduction library [Pou16] for working with greatest fixpoints).

Another view of the entailment relation is as a denotation of **ticl** formulas to predicates over coinductive trees and worlds.

$$\begin{aligned}
\langle _, _ \models_L \varphi \rangle &\in \forall X, \text{ictree}_{E, X} \rightarrow \mathcal{W}_E \rightarrow \mathbb{P} \\
\langle _, _ \models_R \psi_{\mathbf{x}} \rangle &\in \text{ictree}_{E, X} \rightarrow \mathcal{W}_E \rightarrow \mathbb{P}
\end{aligned}$$

By their denotation to predicates, **ticl** formulas form a complete lattice with respect to pointwise implication \Rightarrow_L and \Rightarrow_R in Definition 4.3.1 (shown below) and induce an equivalence relation on formulas (bidirectional implication). Useful (in-)equalities that we proved are shown in Figure 4.12;

$$\begin{array}{c}
\frac{P_X x}{\text{done_with } P_X (\text{Val } x)} \text{DWVAL} \qquad \frac{P_X e \ v \ x}{\text{done_with } P_X (\text{Finish } e \ v \ x)} \text{DWFINISH} \\
\\
\frac{\text{not_done } w \quad P \ w}{\langle t, w \models_L \text{ now } P \rangle} \text{NOW} \models_L \qquad \frac{\langle t, w \models_{L,R} p \rangle \quad \langle t, w \models_{L,R} q \rangle}{\langle t, w \models_{L,R} p \wedge q \rangle} \text{AND} \models \\
\\
\frac{\langle t, w \models_{L,R} p \rangle}{\langle t, w \models_{L,R} p \vee q \rangle} \text{L-OR} \models \qquad \frac{\langle t, w \models_{L,R} p \rangle}{\langle t, w \models_{L,R} p \vee q \rangle} \text{R-OR} \models \\
\\
\frac{\text{anc } \langle t, w \models_L \varphi \rangle \langle t, w \models_L \varphi' \rangle}{\langle t, w \models_L \varphi \ \text{AN} \ \varphi' \rangle} \text{AN} \models_L \qquad \frac{\text{enc } \langle t, w \models_L \varphi \rangle \langle t, w \models_L \varphi' \rangle}{\langle t, w \models_L \varphi \ \text{EN} \ \varphi' \rangle} \text{EN} \models_L \\
\\
\frac{\text{auc } \langle t, w \models_L \varphi \rangle \langle t, w \models_L \varphi' \rangle}{\langle t, w \models_L \varphi \ \text{AU} \ \varphi' \rangle} \text{AU} \models_L \qquad \frac{\text{euc } \langle t, w \models_L \varphi \rangle \langle t, w \models_L \varphi' \rangle}{\langle t, w \models_L \varphi \ \text{EU} \ \varphi' \rangle} \text{EU} \models_L \\
\\
\frac{\text{gfp } (\text{anc } \langle _, _ \models_L \varphi \rangle) \ t \ w}{\langle t, w \models_L \ \text{AG} \ \varphi \rangle} \text{AG} \models_L \qquad \frac{\text{gfp } (\text{enc } \langle _, _ \models_L \varphi \rangle) \ t \ w}{\langle t, w \models_L \ \text{EG} \ \varphi \rangle} \text{EG} \models_L \qquad \frac{\text{done_with } P_X \ w}{\langle t, w \models_R \ \text{done } P_X \rangle} \text{DONE} \models_R \\
\\
\frac{\text{anc } \langle t, w \models_L \varphi \rangle \langle t, w \models_R \psi_X \rangle}{\langle t, w \models_R \varphi \ \text{AN} \ \psi_X \rangle} \text{AN} \models_R \qquad \frac{\text{enc } \langle t, w \models_L \varphi \rangle \langle t, w \models_R \psi_X \rangle}{\langle t, w \models_R \varphi \ \text{EN} \ \psi_X \rangle} \text{EN} \models_R \\
\\
\frac{\text{auc } \langle t, w \models_L \varphi \rangle \langle t, w \models_R \psi_X \rangle}{\langle t, w \models_R \varphi \ \text{AU} \ \psi_X \rangle} \text{AU} \models_R \qquad \frac{\text{euc } \langle t, w \models_L \varphi \rangle \langle t, w \models_R \psi_X \rangle}{\langle t, w \models_R \varphi \ \text{EU} \ \psi_X \rangle} \text{EU} \models_R
\end{array}$$

Figure 4.11: Ticl entailment relations $\models_{L,R}$ by induction on formulas.

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

Figure 4.12: Some of the implications and equivalences in **ticl**. Notation $\Rightarrow_{L,R}$, $\Leftrightarrow_{L,R}$ and formula metavariables p, q capture both prefix and suffix formulas.

not shown are the boolean algebra laws (unit, associativity, commutativity etc) for \wedge, \vee which are also proved in the Coq development.

Definition 4.3.1.

$$\begin{aligned}
\varphi \Rightarrow_L \varphi' &= \forall t, w, \langle t, w \models_L \varphi \rangle \rightarrow \langle t, w \models_L \varphi' \rangle & \varphi \Leftrightarrow_L \varphi' &= \varphi \Rightarrow_L \varphi' \text{ and } \varphi' \Rightarrow_L \varphi \\
\psi_x \Rightarrow_R \psi'_x &= \forall t, w, \langle t, w \models_R \psi_x \rangle \rightarrow \langle t, w \models_R \psi'_x \rangle & \psi_x \Leftrightarrow_R \psi'_x &= \psi_x \Rightarrow_R \psi'_x \text{ and } \psi'_x \Rightarrow_R \psi_x
\end{aligned}$$

Using the (in-)equalities of **ticl** in Figure 4.12 we define the user-facing tactics **cdestruct**, **csplit**, **cleft**, **cright** to step and manipulate **ticl** formulas. Also available, the tactics **cinduction** and **ccoinduction** perform induction on the structure of **AU**, **EU** formulas appearing in the proof context, and coinduction on **AG**, **EG** formulas appearing in the goal. Later in this paper we will introduce many syntax-directed lemmas (Section. 4.4.1) but recognize there are proofs which are only possible by low-level induction and coinduction. The **cinduction** and **ccoinduction** tactics serve as a “trap-door” for low-level proofs when needed.

$\models_{L,R}$ is \sim **invariant**

Although the transition relation $\mapsto \in \text{relation}(\text{ictree}_{E, X} * \mathcal{W}_E)$ is not invariant to with respect to *up-to-guard* (\sim) equivalence, we prove both notions of **ticl** entailment ($\models_{L,R}$) are \sim -invariant. This result enables rewriting with the **ictree** equational theory (Figure 4.4) on the left-side of

entailment relations $\models_{L,R}$. Invariance to \sim also allows erasing a finite number of **Guard** constructors, unfolding loops and simplifying monadic computation, all of which are used to verify the examples in Section 4.5.

4.3.4. Coinductive Proofs and Up-to Principles in Coq

We briefly focus on coinduction—the *always* operators **AG**, **EG** in **ticl** require defining several coinductive relations and proofs. We rely on the **coinduction** library [Pou16] to define greatest fixpoints over the complete lattice of Coq propositions. Note: this section is aimed at the reader interested in understanding the internals of our library, it can be safely skipped at first read.

The primary construction offered by the **coinduction** library is a greatest fixpoint operator (**gfp** $b : X$) for any complete lattice X and monotone endofunction $b : X \rightarrow X$. Specifically, the library proves Coq 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 **ictree** and worlds so we work in the complete lattice $(\mathbf{ictree}_{E, X} \rightarrow \mathcal{W}_E \rightarrow \mathbb{P})$.

The **coinduction** library 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 $f : X \rightarrow X$ 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 4.13 we give the up-to-principles for coinduction proofs in **ticl**. The $\text{UPTO}^{UP}(\text{equiv})$ principle is used to show $\text{UPTO}^{UP}(\text{equiv}) \leq \lambda t. \mathbf{gfp}(\mathbf{anc} \varphi) t$, meaning equivalent trees (abstracting over the exact equivalence relation) satisfy the same **AG** φ formula (similarly **EG** φ). Note, $\text{UPTO}^{UP}(\text{equiv})$ is sufficiently general; any equivalence relation **equiv** satisfying the **EXEQUIV** lemma in Figure 4.8 can be used.

$$\begin{aligned}
\text{UPTO}^{UP}(\text{equiv}) \mathcal{R} &\triangleq \{t \mid \exists t', \text{equiv } t t' \wedge \mathcal{R} t'\} \\
\text{BINDAG}^{UP}(\varphi, P_X) \mathcal{R} &\triangleq \{(t \gg k, w) \mid \langle t, w \models_R \varphi \text{ AU AX done } P_X \rangle \\
&\quad \wedge (\forall x, w, P_X x w \rightarrow \mathcal{R} (k x) w)\} \\
\text{BINDEG}^{UP}(\varphi, P_X) \mathcal{R} &\triangleq \{(t \gg k, w) \mid \langle t, w \models_R \varphi \text{ EU EX done } P_X \rangle \\
&\quad \wedge (\forall x, w, P_X x w \rightarrow \mathcal{R} (k x) w)\}
\end{aligned}$$

Figure 4.13: Up-to-principles for coinductive AG, EG proofs.

Up-to-principles $\text{BINDAG}^{UP}(\varphi, P_X)$, $\text{BINDEG}^{UP}(\varphi, P_X)$, parametrized by a prefix formula φ and a postcondition $P_X \in X \rightarrow \mathcal{W}_E \rightarrow \mathbb{P}$ are used to prove the bind lemmas in Figure 4.15. Specifically, by showing the bind principle is under the greatest fixpoint $\text{BINDAG}^{UP}(\varphi, P_X) \leq \text{gfp}(\text{anc } \varphi)$ we reduce a coinductive proof $\langle x \leftarrow \mathbf{t};; \mathbf{k} x, w \models_L \text{AG } \varphi \rangle$ to an *inductive* proof on the finite prefix $\langle t, w \models_R \varphi \text{ AU AX done } P_X \rangle$ and a coinductive proof about its continuation k .

4.4. Structural lemmas

In this section we propose the structural temporal logic lemmas of **ticl** and show it is possible to write proofs in a high-level of abstraction. The lemmas in this section *internalize* low-level (co-)inductive principles to simple lemmas about sequential composition, conditionals and loops, allowing for the compositional reasoning of liveness properties.

We proceed in two phases; in the first phase define and prove structural, temporal logic lemmas over general **ictree** models and their combinators (\oplus , \gg , **iter**). In the second phase, we define a small stateful, imperative language **StImp** with assignment, conditionals, loops and nondeterminism and give specialized versions of the structural lemmas for instrumented programs written in the **StImp** language. Extensions to the language **StImp** with queues, concurrency and message-passing are then used to verify the examples in the next section.

4.4.1. Structural rules for **ictree**

The table in Figure. 4.14 shows the cartesian product of **ictree** combinators and **ticl** modal operators. We have identified and proved backward-reasoning lemmas (\Leftarrow) for the basic **ictree** combinators and bidirectional lemmas (\Leftrightarrow) for the **ictree** constructors and \emptyset . We conjecture there

are useful inversion lemmas for **bind** and **iter** as well, which we leave for future work.

	Prefix (φ)						Suffix (ψ_x)			
	AN	EN	AU	EU	AG	EG	AN	EN	AU	EU
Ret	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow
Br	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow
Vis	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow
\emptyset	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow
\gg	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow
iter	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow

Figure 4.14: Library of structural, compositional lemmas for **ictree** combinators and **ticl** operators. Symbol \Leftarrow indicates a backward-reasoning lemma and \Leftrightarrow lemmas in both directions.

Sequential composition

$$\begin{array}{c}
\frac{\langle t, w \models_L \varphi \rangle}{\langle x \leftarrow t;; k \ x, w \models_L \varphi \rangle} \text{BINDL} \quad \frac{\langle t \oplus u, w \models_L \varphi \rangle \quad \langle t, w \models_L \varphi \text{ AU } \varphi' \rangle \quad \langle u, w \models_L \varphi \text{ AU } \varphi' \rangle}{\langle t \oplus u, w \models_L \varphi \text{ AU } \varphi' \rangle} \text{BRAU}_L \\
\\
\frac{\langle t \oplus u, w \models_L \varphi \rangle \quad \langle t, w \models_L \varphi \text{ EU } \varphi' \rangle \vee \langle u, w \models_L \varphi \text{ EU } \varphi' \rangle}{\langle t \oplus u, w \models_L \varphi \text{ EU } \varphi' \rangle} \text{BREU}_L \\
\\
\frac{\langle t, w \models_R \varphi \text{ AU AX done } \mathcal{R}_Y \rangle \quad \forall y, w, \mathcal{R}_Y y w \rightarrow \langle k \ y, w \models_L \varphi \text{ AU } \varphi' \rangle}{\langle x \leftarrow t;; k \ x, w \models_L \varphi \text{ AU } \varphi' \rangle} \text{BINDAU}_L \\
\\
\frac{\langle t, w \models_R \varphi \text{ AU AX done} = y \ w' \rangle \quad \langle k \ y, w' \models_L \varphi \text{ AU } \varphi' \rangle}{\langle x \leftarrow t;; k \ x, w \models_L \varphi \text{ AU } \varphi' \rangle} \text{BINDAU}_{L=} \\
\\
\frac{\langle t, w \models_R \varphi \text{ AU AX done } \mathcal{R}_Y \rangle \quad \forall y, w, \mathcal{R}_Y y w \rightarrow \langle k \ y, w \models_R \varphi \text{ AU } \psi'_x \rangle}{\langle x \leftarrow t;; k \ x, w \models_R \varphi \text{ AU } \psi'_x \rangle} \text{BINDAU}_R \\
\\
\frac{\langle t, w \models_R \varphi \text{ EU EX done} = y \ w' \rangle \quad \langle k \ y, w' \models_R \varphi \text{ EU } \psi_x \rangle}{\langle x \leftarrow t;; k \ x, w \models_R \varphi \text{ EU } \psi_x \rangle} \text{BINDEU}_R \\
\\
\frac{\langle t, w \models_R \varphi \text{ AU AX done } \mathcal{R}_Y \rangle \quad \forall y, w, \mathcal{R}_Y y w \rightarrow \langle k \ y, w \models_L \text{ AG } \varphi \rangle}{\langle x \leftarrow t;; k \ x, w \models_L \text{ AG } \varphi \rangle} \text{BINDAG}
\end{array}$$

Figure 4.15: Representative structural lemmas for nondeterminism and sequential composition of **ictree**_{E, X}.

Sequential program composition is implemented through Monad combinators (**Ret**, \gg) in **itrees**. The structural lemmas in Figure 4.15 split temporal specifications of sequential programs into modular subproofs, similar to the sequence rule from Hoare logic. For example, if $x \leftarrow t;; k \ x$ is a command-line application and the goal is to prove it will *eventually* print to the terminal, that is:

$\langle x \leftarrow t;; k\ x, w \models_L \text{AF obs PRINTS} \rangle$. There are two possibilities:

1. Either t prints, use the BINDL lemma to show it and ignore the continuation k .
2. Or the continuation k prints. Use the BINDAU_L lemma to show t terminates with some postcondition on return values and worlds \mathcal{R}_Y . There could be more than one possible return values if t is nondeterministic. Then for all possible return values $y \in Y$ and worlds $w' \in \mathcal{W}_E$ such that $\mathcal{R}_Y\ y\ w'$, we must show the continuation $k\ y$ eventually prints to terminal $\langle k\ y, w' \models_L \text{AF obs PRINTS} \rangle$.

In their general form the BINDAU_L, BINDAU_R lemmas can be cumbersome to apply as they require manually specifying the postcondition \mathcal{R}_Y . In practice, for deterministic programs we can rely on Coq's existential variables (**evvars**) to postpone instantiation of the return value to automatic *unification*. The convenience lemma BINDAU_{L=} assumes a finite, linear path exists so eventually a single return value and world will be reached.

Iteration

The loop combinator ($\text{iter} \in (I \rightarrow \text{ictree}_{E, I+R}) \rightarrow I \rightarrow \text{ictree}_{E, R}$) is capable of expressing both terminating and non-terminating loops, depending on the result of the stepping function ($\text{step} \in I \rightarrow \text{ictree}_{E, I+R}$). If **step** returns the left-injection of type **I** (iterator), the loop continues and the step function will be called again with the new iterator. If the loop returns the right-injection of type **R** (result) the loop terminates, returning the result. In Figure 4.16, we provide lemmas to prove both loop *termination* and loop *invariance* for finite and infinite loops respectively.

Rule ITERAU_L in Figure 4.16 is an *eventually* lemma. A relation $\mathcal{R} \in I \rightarrow \mathcal{W}_E \rightarrow \text{Prop}$ must be specified called the *loop invariant*, as well as a binary relation $\mathcal{R}_v \in \text{relation}(I * \mathcal{W}_E)$ called the *loop variant*. Invariant \mathcal{R} appears on both sides of the implication in the premise of the rule ITERAU_L, so it must be picked carefully to encapsulate the program state before and after the loop body. The relation \mathcal{R}_v , contrary to the invariant, describes how the program's state *evolves* over time. To ensure termination \mathcal{R}_v must be *well-founded*, meaning there are no infinite \mathcal{R}_v chains.

Working with well-founded relations in Coq directly can be difficult, so we define the simplified rule $\text{ITERAU}_{L,\mathbb{N}}$ that expects a function to the natural numbers ($f \in I \rightarrow \mathcal{W}_E \rightarrow \mathbb{N}$), such that successive pairs of iterator and world are strictly monotonically decreasing. Functions like f are sometimes called *ranking functions* and finding suitable ranking functions can be challenging; recent work on automatic inference of ranking functions [YTGN24] applies, if a suitable ranking function is inferred, then rule $\text{ITERAU}_{L,\mathbb{N}}$ can help verify program termination.

Delving into the body of rule ITERAU_L , the main premise of the rule is split in two cases; the first is the *base case* of the underlying induction ($\langle \mathbf{k} \ i, w \models_L \varphi \ \mathbf{AU} \ \varphi' \rangle$) and the second case is the *inductive step*. In the base case, if we prove the loop body satisfies the condition $\langle k \ i, w \models_L \varphi \ \mathbf{AU} \ \varphi' \rangle$ (φ until eventually φ') then the whole loop also satisfies $\varphi \ \mathbf{AU} \ \varphi'$, we are done. In the *inductive step* case of ITERAU_L , we must show that when the loop does not terminate ($lr = \mathbf{inl} \ i'$), the new iterator (i') satisfies the invariant \mathcal{R} and the variant \mathcal{R}_v shows it is part of a decreasing finite chain.

Continuing with the *termination* rule ITERAU_R , this is the suffix formula equivalent rule to ITERAU_L . Rule ITERAU_L expects formula φ' to be eventually satisfied, even if the loop keeps running afterwards. Termination rule ITERAU_R expects the loop to terminate with a value and world satisfying $\psi_{\mathbf{x}}$. In the premises of ITERAU_R there are two cases, if $lr = \mathbf{inl} \ i'$ then this is the same as ITERAU_L the invariant $\mathcal{R} \ i' \ w'$ and variant $\mathcal{R}_v \ (i', w') \ (i, w)$ must be satisfied prior to loop re-entry. The second case concludes the proof, by showing that eventually the loop will exit with $lr = \mathbf{inr} \ r$, then $\langle \mathbf{Ret} \ r, w \models_R \varphi \ \mathbf{AN} \ \psi_{\mathbf{x}} \rangle$ where $\psi_{\mathbf{x}}$ is the loop postcondition.

Finally let's explore the behavior of *nonterminating* loops with the *invariance* rule ITERAG . *Always* and *always-eventually* properties can be proved by invariance. The premise of the rule requires specifying a loop invariant \mathcal{R} on iterators and worlds, similar to rule ITERAU_L . In the rule premise there are two conditions that must hold *for every iteration*. In the first premise $\langle \mathbf{iter} \ \mathbf{k} \ i, w \models_L \varphi \rangle$, we will return to this premise shortly. In the second premise, we get the loop body $k \ i$ must satisfy φ until it terminates, and then the iterator $lr = \mathbf{inl} \ i'$ and world w' must satisfy the loop invariant $\mathcal{R} \ i \ w$. The second premise of the *invariance* rule enforces the loop body must terminate, of course this is not always the case, for example we can have two nested infinite loops. In that case, we recall

$$\begin{array}{c}
\mathcal{R} \, i \, w \quad \text{well_founded } \mathcal{R}_v \\
\forall i, w, \mathcal{R} \, i \, w \rightarrow \\
\langle \mathbf{k} \, i, w \models_L \varphi \, \mathbf{AU} \, \varphi' \rangle \vee \\
\langle \mathbf{k} \, i, w \models_R \varphi \, \mathbf{AU} \, \mathbf{AX} \, \text{done} \, (\lambda \, lr \, w' \Rightarrow \\
\quad \exists i', lr = \mathbf{inl} \, i' \wedge \mathcal{R} \, i' \, w' \\
\quad \wedge \mathcal{R}_v \, (i', w') \, (i, w)) \rangle \\
\hline
\langle \mathbf{iter} \, \mathbf{k} \, i, w \models_L \varphi \, \mathbf{AU} \, \varphi' \rangle \quad \text{ITERAU}_L
\end{array}
\quad
\begin{array}{c}
\mathcal{R} \, i \, w \\
\forall i, w, \mathcal{R} \, i \, w \rightarrow \\
\langle \mathbf{k} \, i, w \models_L \varphi \, \mathbf{AU} \, \varphi' \rangle \vee \\
\langle \mathbf{k} \, i, w \models_R \varphi \, \mathbf{AU} \, \mathbf{AX} \, \text{done} \, (\lambda \, lr \, w' \Rightarrow \\
\quad \exists i', lr = \mathbf{inl} \, i' \wedge \mathcal{R} \, i' \, w' \\
\quad \wedge f \, i' \, w' < f \, i \, w) \rangle \\
\hline
\langle \mathbf{iter} \, \mathbf{k} \, i, w \models_L \varphi \, \mathbf{AU} \, \varphi' \rangle \quad \text{ITERAU}_{L, \mathbb{N}}
\end{array}$$

$$\begin{array}{c}
\mathcal{R} \, i \, w \quad \text{well_founded } \mathcal{R}_v \\
\forall i, w, \mathcal{R} \, i \, w \rightarrow \\
\langle \mathbf{k} \, i, w \models_R \varphi \, \mathbf{AU} \, \mathbf{AX} \, \text{done} \, (\lambda \, lr \, w' \Rightarrow \\
\quad \begin{cases} \mathcal{R} \, i' \, w' \wedge \mathcal{R}_v \, (i', w') \, (i, w), & \text{if } lr = \mathbf{inl} \, i' \\
\langle \mathbf{Ret} \, r, w' \models_R \varphi \, \mathbf{AN} \, \psi_x \rangle, & \text{if } lr = \mathbf{inr} \, r \end{cases} \\
\quad) \rangle \\
\hline
\langle \mathbf{iter} \, \mathbf{k} \, i, w \models_R \varphi \, \mathbf{AU} \, \psi_x \rangle \quad \text{ITERAU}_R
\end{array}
\quad
\begin{array}{c}
\mathcal{R} \, i \, w \\
\forall i, w, \mathcal{R} \, i \, w \rightarrow \\
\langle \mathbf{iter} \, \mathbf{k} \, i, w \models_L \varphi \rangle \wedge \\
\langle \mathbf{k} \, i, w \models_R \mathbf{AX}(\varphi \, \mathbf{AU} \, \mathbf{AX} \, \text{done} \, (\lambda \, lr \, w' \Rightarrow \\
\quad \exists i', lr = \mathbf{inl} \, i' \wedge \mathcal{R} \, i' \, w')) \rangle \\
\hline
\langle \mathbf{iter} \, \mathbf{k} \, i, w \models_L \mathbf{AG} \, \varphi \rangle \quad \text{ITERAG}
\end{array}$$

Figure 4.16: Representative iteration lemmas for **AU**, **AG** and $\mathbf{ictree}_{E, X}$.

iter is defined in terms of monadic bind and the **BINDL** rule applies, then it suffices to show the inner loop satisfies the invariance lemma.

$\mathbf{AExp} \in \text{Type}$
 $\mathbf{AExp} = \mid \text{var } (s \in \text{string}) \mid \text{val } (n \in \mathbb{N}) \mid (x \in \mathbf{AExp}) + (y \in \mathbf{AExp}) \mid (x \in \mathbf{AExp}) - (y \in \mathbf{AExp})$

$\mathbf{BExp} \in \text{Type}$
 $\mathbf{BExp} = \mid (x \in \mathbf{AExp}) = (y \in \mathbf{AExp}) \mid (x \in \mathbf{AExp}) < (y \in \mathbf{AExp}) \mid \mathbf{true}$
 $\mid (x \in \mathbf{BExp}) \wedge (y \in \mathbf{BExp}) \mid (x \in \mathbf{BExp}) \vee (y \in \mathbf{BExp}) \mid \mathbf{false}$

$\mathbf{StImp} \in \text{Type}$
 $\mathbf{StImp} = \mid (s \in \text{string}) \leftarrow (y \in \mathbf{AExp}) \mid \text{if } (c \in \mathbf{BExp}) \text{ then } x \in \mathbf{StImp} \text{ else } y \in \mathbf{StImp}$
 $\mid (l \in \mathbf{StImp}) ; (r \in \mathbf{StImp}) \mid \text{do } b \in \mathbf{StImp} \text{ while } (c \in \mathbf{BExp})$
 $\mid (l \in \mathbf{StImp}) \oplus (r \in \mathbf{StImp}) \mid \mathbf{skip}$

Figure 4.17: Syntax of a small imperative language **StImp** with mutable state and nondeterminism.

$\mathcal{M} \in \text{Type} = \text{Map}_{\text{string}, \mathbb{N}}$
 $\cup \in \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M}$ (map union)
 $(s \in \text{string} \hookrightarrow n \in \mathbb{N}) \in \mathcal{M}$ (singleton map)
 $(m \in \mathcal{M})[s \in \text{string}] \in \mathbb{N}$ (total get)

Figure 4.18: Some auxiliary map operations from Coq's standard library are assumed.

Returning to the first premise of the *invariance* lemma **ITERAG**, it might seem unnecessary at first

to enforce $\langle \text{iter } k \ i, w \models_L \varphi \rangle$ since the second premise of the rule also enforces φ until termination of the loop body ($k \ i$). However, recall the **rotate** example in Figure 4.1 and the *always-eventually* specification. Program **rotate** is a single loop, omitting premise $\langle \text{iter } k \ i, w \models_L \varphi \rangle$ means we have to show the loop body individually satisfies the condition “ x is eventually observed in the head position” but the loop body by itself cannot satisfy the “eventually” as it takes multiple loop iterations for the “eventually” to happen. Hence preserving the **iter** loop in the first premise is necessary to prove *always-eventually* properties. What is noteworthy about the *invariance* lemma ITERAG is it can discharge a coinductive goal to two subgoals, none of which necessarily requires coinduction to prove, thus internalizing coinductive proofs to a simple rule application.

$$\begin{aligned}
\text{state}_{\mathcal{M}} &\in \text{Type} \rightarrow \text{Type} = \mid (\text{Get} \in \text{state}_{\mathcal{M},\mathcal{M}}) \mid (\text{Put } (m \in \mathcal{M}) \in \text{state}_{\mathcal{M},\text{unit}}) \\
h_{\text{state}_{\mathcal{M}}} &\in \text{state}_{\mathcal{M}} \rightsquigarrow \text{InstrM}_{\mathcal{M},\mathcal{M}} \\
h_{\text{state}_{\mathcal{M}}} (\text{Get} \in \text{state}_{\mathcal{M},\mathcal{M}}) (m \in \mathcal{M}) &= \text{Ret } (m, m) \\
h_{\text{state}_{\mathcal{M}}} (\text{Put } m' \in \text{state}_{\mathcal{M},\text{unit}}) (_ \in \mathcal{M}) &= \text{Vis } (\text{Log } m') (\lambda (_ \in \text{unit}) \Rightarrow \text{Ret } ((), m')) \\
\llbracket _ \rrbracket_{A,_} &\in \text{AExp} \rightarrow \mathcal{M} \rightarrow \mathbb{N} \\
\llbracket \text{var } s \rrbracket_{A,m} &= m[s], \quad \llbracket x + y \rrbracket_{A,m} = \llbracket x \rrbracket_{A,m} + \llbracket y \rrbracket_{A,m} \\
\llbracket \text{val } n \rrbracket_{A,_} &= n, \quad \llbracket x - y \rrbracket_{A,m} = \llbracket x \rrbracket_{A,m} - \llbracket y \rrbracket_{A,m} \\
\llbracket _ \rrbracket_{B,_} &\in \text{BExp} \rightarrow \mathcal{M} \rightarrow \mathbb{B} \\
\llbracket x = y \rrbracket_{B,m} &= \llbracket x \rrbracket_{A,m} == \llbracket y \rrbracket_{A,m} \\
\llbracket x < y \rrbracket_{B,m} &= \llbracket x \rrbracket_{A,m} < \llbracket y \rrbracket_{A,m} \\
\llbracket a \wedge b \rrbracket_{B,m} &= \llbracket a \rrbracket_{B,m} \&\& \llbracket b \rrbracket_{B,m} \\
\llbracket a \vee b \rrbracket_{B,m} &= \llbracket a \rrbracket_{B,m} \parallel \llbracket b \rrbracket_{B,m} \\
\llbracket _ \rrbracket &\in \text{StImp} \rightarrow \text{ictree}_{\text{state}_{\mathcal{M}}, \text{unit}} \\
\llbracket s \leftarrow x \rrbracket &= \text{get} \gg \lambda m \Rightarrow \text{put } ((s \hookrightarrow \llbracket x \rrbracket_{A,m}) \cup m), \quad \llbracket t ; u \rrbracket = \llbracket t \rrbracket ; \llbracket u \rrbracket \\
\llbracket \text{skip} \rrbracket &= \text{Ret } (), \quad \llbracket t \oplus u \rrbracket = \llbracket t \rrbracket \oplus \llbracket u \rrbracket \\
\llbracket \text{if } (c) \text{ then } t \text{ else } u \rrbracket &= \text{get} \gg \lambda m \Rightarrow \begin{cases} \llbracket t \rrbracket, & \text{if } \llbracket c \rrbracket_{B,m} \\ \llbracket u \rrbracket, & \text{otherwise} \end{cases} \\
\llbracket \text{do } t \text{ while } (c) \rrbracket &= \text{iter} \left(\lambda (_ \in \text{unit}) \Rightarrow \llbracket t \rrbracket ; \text{get} \gg \lambda m' \Rightarrow \begin{cases} \text{Ret } (\text{inl } ()), & \text{if } \llbracket c \rrbracket_{B,m'} \\ \text{Ret } (\text{inr } ()), & \text{otherwise} \end{cases} \right) ()
\end{aligned}$$

$$\mathcal{J}[\llbracket t \rrbracket \in \text{StImp}]_{(m \in \mathcal{M})} \in \text{InstrM}_{\mathcal{M},\mathcal{M}} = \text{instr } h_{\text{state}_{\mathcal{M}}} \llbracket t \rrbracket m$$

Figure 4.19: Denotation of **StImp** programs to the instrumentation monad $\text{InstrM}_{\mathcal{M},\mathcal{M}}$ by intermediate interpretation to $\text{ictree}_{\text{state}_{\mathcal{M}}, \text{unit}}$.

4.4.2. Structural rules for **StImp**

First we define the syntax of the small imperative language **StImp** with mutable state and non-determinism in Figure 4.17. The semantics of **StImp** are defined in terms of $\mathbf{ictree}_{\mathbf{state}_{\mathcal{M}}, \mathbf{unit}}$ in Figure 4.19 where $\mathbf{state}_{\mathcal{M}}$ is the type of events over a mutable shared heap **string** indexing and \mathbb{N} values. Low-level operations on maps are assumed from Coq’s standard library (Figure 4.18); $m_1 \cup m_2$ is map union, $s \mapsto x$ is the singleton map with key s and value x , $m[s]$ is the total “get” that returns the value associated with s or 0 if it does not exist. Finally $\mathcal{J}[\![t]\!]_m$ performs *instrumented evaluation* of the **StImp** program t with an initial state ($m \in \mathcal{M}$). The instrumentation handler $h_{\mathbf{state}_{\mathcal{M}}}$ records the entire state on *put* events and erases *get* events. Further extensions to the instrumentation handler with additional *ghost-state* are possible without much change in the structural rules.

Equipped with instrumentation of **StImp** programs to the $\mathbf{InstrM}_{\mathcal{M}, \mathcal{M}}$ monad, we proceed to lift the $\mathbf{ictree}_{E, X}$ structural rules of Figures 4.15, 4.16 over to the **StImp** language, recalling that the instrumentation monad is yet another $\mathbf{ictree}_{\log_{\mathcal{M}}, \mathbf{unit}}$. Hence we can plug instrumented programs ($\mathcal{J}[\![t]\!]_m$) in the left-hand side of the entailment relations $\models_{L,R}$ and reason about temporal formulas over states (\mathcal{M}). A few representative structural lemmas for assignment, sequential composition, conditionals non-determinism and iteration are given in Figure 4.20 with respect to the temporal operators **AU**, **AG**. The full array of program structures and temporal operators is proven in our Coq development and omitted here in the interest of space.

The structural rules for **StImp** are backwards reasoning, the goal is in the bottom and proof obligations are given on the top of the inference line. The proof obligations generated are “smaller” than the goal they apply. Either the proof obligation refers to subprogram of the program in the goal, for example rules $\mathbf{STSEQ}_L, \mathbf{STIF}_{\top} \mathbf{AU}_L, \mathbf{STIF}_{\perp} \mathbf{AU}_L, \mathbf{STSEQ}_{\mathbf{AG}}, \mathbf{STITERA}_{\mathbf{AU}_R}, \mathbf{STITERA}_{\mathbf{AU}_L}, \mathbf{N}$, or the proof obligation formula is a subformula of the one in the goal, for example in the *invariance* rule $\mathbf{STITERA}_{\mathbf{AG}}$ in the first proof obligation φ is a syntactic subformula of $\mathbf{AG} \varphi$.

In the next section we proceed to extend **StImp** with queue operations (**pushx**, **pop**), secure shared

$$\begin{array}{c}
\frac{\langle \mathcal{J}[s \leftarrow x]_m, w \models_L \varphi \rangle \quad m' = (s \hookrightarrow \llbracket x \rrbracket_{A,m}) \cup m \quad \langle \mathcal{J}[\mathbf{skip}]_{m'}, \mathbf{Obs}(\mathbf{Log} m') () \models_L \varphi' \rangle}{\langle \mathcal{J}[s \leftarrow x]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle} \text{STASSIGNAU}_L \\
\\
\frac{\langle \mathcal{J}[t]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle}{\langle \mathcal{J}[t ; u]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle} \text{STSEQ}_L \quad \frac{\langle \mathcal{J}[t]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle \quad \llbracket c \rrbracket_{B,m}}{\langle \mathcal{J}[\mathbf{if} (c) \text{ then } t \text{ else } u]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle} \text{STIF}_\top \text{AU}_L \\
\frac{\langle \mathcal{J}[u]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle \quad \neg \llbracket c \rrbracket_{B,m}}{\langle \mathcal{J}[\mathbf{if} (c) \text{ then } t \text{ else } u]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle} \text{STIF}_\perp \text{AU}_L \\
\frac{\langle \mathcal{J}[t \oplus u]_m, w \models_L \varphi \rangle \quad \langle \mathcal{J}[t]_m, w \models_R \varphi \mathbf{AU} \psi_{\text{unit}} \rangle \quad \langle \mathcal{J}[u]_m, w \models_R \varphi \mathbf{AU} \psi_{\text{unit}} \rangle}{\langle \mathcal{J}[t \oplus u]_m, w \models_R \varphi \mathbf{AU} \psi_{\text{unit}} \rangle} \text{STBRAU}_R \\
\\
\frac{\langle \mathcal{J}[t]_m, w \models_R \varphi \mathbf{AU} \mathbf{AX done} = ((), m') w' \rangle \quad \langle \mathcal{J}[u]_m, w \models_R \varphi \mathbf{AU} \psi_{\text{unit}} \rangle}{\langle \mathcal{J}[t ; u]_m, w \models_R \varphi \mathbf{AU} \psi_{\text{unit}} \rangle} \text{STSEQAU}_{R=} \\
\\
\frac{\langle \mathcal{J}[t]_m, w \models_R \varphi \mathbf{AU} \mathbf{AX done} \mathcal{R} \rangle \quad (\forall m', w', \mathcal{R} m' w' \rightarrow \langle \mathcal{J}[u]_{m'}, w' \models_L \mathbf{AG} \varphi \rangle)}{\langle \mathcal{J}[t ; u]_m, w \models_L \mathbf{AG} \varphi \rangle} \text{STSEQAG} \\
\\
\frac{\mathcal{R} m w \quad \text{well_founded } \mathcal{R}_v \quad \forall m, w, \mathcal{R} m w \rightarrow \quad \langle \mathcal{J}[t]_m, w \models_R \varphi \mathbf{AU} \mathbf{AX done} (\lambda '(_, m') w' \Rightarrow \begin{cases} \mathcal{R} m' w' \wedge \mathcal{R}_v(ctx', w')(ctx, w), & \text{if } \llbracket c \rrbracket_{B,m'} \\ \langle \mathcal{J}[\mathbf{skip}]_m, w' \models_R \varphi \mathbf{AN} \psi_{\text{unit}} \rangle, & \text{otherwise} \end{cases})}{\langle \mathcal{J}[\mathbf{do } t \text{ while } (c)]_m, w \models_R \varphi \mathbf{AU} \psi_{\text{unit}} \rangle} \text{STWHILEAU}_R \\
\\
\frac{\mathcal{R} m w \quad (f \in \mathcal{M} \rightarrow \mathbb{N}) \quad \forall m, w, \mathcal{R} m w \rightarrow \quad \langle \mathcal{J}[t]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle \vee \quad \langle \mathcal{J}[t]_m, w \models_R \varphi \mathbf{AU} \mathbf{AX done} (\lambda '(_, m') w' \Rightarrow \llbracket c \rrbracket_{B,m'} \wedge \mathcal{R} m' w' \wedge f m' < f m)}{\langle \mathcal{J}[\mathbf{do } t \text{ while } (c)]_m, w \models_L \varphi \mathbf{AU} \varphi' \rangle} \text{STWHILEAU}_{L,\mathbb{N}} \\
\\
\frac{\mathcal{R} m w \quad \forall m, w, \mathcal{R} m w \rightarrow \quad \langle \mathcal{J}[\mathbf{do } t \text{ while } (c)]_m, w \models_L \varphi \rangle \wedge \quad \langle \mathcal{J}[t]_m, w \models_R \mathbf{AX}(\varphi \mathbf{AU} \mathbf{AX done} (\lambda '(_, m') w' \Rightarrow \llbracket c \rrbracket_{B,m'} \wedge \mathcal{R} m' w'))}{\langle \mathcal{J}[\mathbf{do } t \text{ while } (c)]_m, w \models_L \mathbf{AG} \varphi \rangle} \text{STWHILEAG}
\end{array}$$

Figure 4.20: Representative structural lemmas for language **StImp** and **ticl** operators **AU**, **AG**.

state, and message passing operations. We then use the lemmas in Figure 4.20 to structurally prove both coinductive and inductive properties like invariance and termination, as well as nested *always-eventually* properties. No explicit use of the `induction` or `coinduction` tactics is used anywhere in our examples.

4.5. Motivating examples

We evaluated `ticl` by structurally verifying several examples from the T2 CTL benchmark suite [BCI⁺16] and on four examples inspired from computer systems; two programs on queues, a secure shared memory program, and a distributed consensus protocol. For each one, we extend the imperative language `StImp` with additional effects and define new instrumentation handlers to observe these effects.

4.5.1. Queues

$$\begin{aligned}
\mathcal{Q} \in \text{Type} &= \text{list } \mathbb{N} \\
E_{\mathcal{Q}} \in \text{Type} \rightarrow \text{Type} &= \mid (\text{Push } (x \in \mathcal{M}) \in E_{\mathcal{Q}} \text{ unit}) \mid (\text{Pop} \in E_{\mathcal{Q}} \mathbb{N}) \\
\text{AExp}_{\mathcal{Q}} \in \text{Type} &= \mid \text{var } (s \in \text{string}) \mid \dots \mid \text{pop} \\
\text{BExp}_{\mathcal{Q}} \in \text{Type} &= \mid (x \in \text{AExp}_{\mathcal{Q}}) = (y \in \text{AExp}_{\mathcal{Q}}) \mid \dots \\
\text{StImp}_{\mathcal{Q}} \in \text{Type} &= \mid (s \in \text{string}) \leftarrow (y \in \text{AExp}_{\mathcal{Q}}) \mid \dots \mid \text{push } (x \in \text{AExp}_{\mathcal{Q}}) \\
h_{\mathcal{Q}} \in E_{\mathcal{Q}} \rightsquigarrow \text{InstrM}_{\mathcal{Q}, \mathbb{N}} \\
h_{\mathcal{Q}} (\text{Push } x \in E_{\mathcal{Q}} \text{ unit}) (q \in \mathcal{Q}) &= \text{Ret } ((), x ++ q) \\
h_{\mathcal{Q}} (\text{Pop} \in E_{\mathcal{Q}} \mathbb{N}) (h::ts \in \mathcal{Q}) &= \text{Vis } (\text{Log } h) (\lambda _ \in \text{unit}) \Rightarrow \text{Ret } (h, ts) \\
h_{\mathcal{Q}} (\text{Pop} \in E_{\mathcal{Q}} \mathbb{N}) ([\] \in \mathcal{Q}) &= \text{Ret } (0, [\])
\end{aligned}$$

Figure 4.21: Language `StImpQ` extends the language `StImp` with a global queue as additional state, operations `push` and `pop` interact with the queue.

We start with the language of queues `StImpQ` in Figure 4.21 and two nonterminating programs: `drain` (Figure 4.22) and `rotate` (Figure 4.1). We instrument the programs using the queue instrumentation handler $h_{\mathcal{Q}}$ to define queue instrumentation `instrQ`. The process of giving instrumenting semantics to new events is similar to effect handlers in similar to our `StImp` definitions ($\mathcal{J}[t]_m$).

`Definition drain :=` `Theorem drain_af: $\forall(x: T) (q: \text{list } T),$`
`do (x \leftarrow pop) while (true)` `<(instrQ drain (q ++ [x])), Pure |= AF obs (λ hd \Rightarrow hd = x)>.`

Figure 4.22: Program `drain` runs forever, pops all elements in the queue until it eventually spins on the empty queue. Specification `drain_af` is; eventually element `x` will be observed in the head of the queue.

Queue drain (eventually)

For `drain`, the target specification is an *eventually* (AF) property, where AF is syntactic notation for $\top \text{ AU}$. The proof proceeds by backwards reasoning, starting from the goal in the bottom of Figure 4.23 and applying `tic1` lemmas and Coq tactics upwards. Using $\text{STITERAU}_{L,\mathbb{N}}$ we “enter” the loop body, by specifying the *loop invariant* `Rinv` and ranking function `length`. Even though the `drain` program is infinite, each iteration emits a monotonically decreasing series of queues, until it reaches the empty queue.

The loop invariant `Rinv` is defined by case analysis on the world w . When $w = \text{Pure}$ no element has been removed yet—the program just started. When $w = \text{Obs } h' \text{ tt}$ the most recent element popped from the queue is h' . If the queue is empty, then h' must have been the last element in the queue, so the goal property $x = h'$ must be satisfied. If the queue is not empty there must be some finite prefix hs left to drain before reaching x .

The proof in Figure 4.23 proceeds by applying `tic1` lemmas and low-level Coq tactics like `destruct`, but implicitly this is a proof by induction on the length of the queue q . We never have to invoke the `induction` tactic, it is silently applied in the proof of $\text{STWHILEAU}_{L,\mathbb{N}}$. Implicit induction and coinduction are the most appealing aspect of `tic1`; complex reasoning about very general computation structures and formulas is internalized in a way that is opaque to the user of the logic. With respect to mechanization, a valid loop invariant `Rinv` and ranking function `f` are necessary, as is the case in most Program Logics, and there is potential for proof automation as the rest of the proof is syntax-driven, by the syntax of formulas and programs.

Queue rotate (always-eventually)

The second program in the language of queues we encountered early on; **rotate** from Figure 4.1. Unlike **drain** which is guaranteed to empty the queue, rotate re-pushes elements in the back of the queue and, like **drain**, it runs forever.

The target specification is the *always-eventually* property **rotate_agaf** in Figure 4.1. We prove this property by application of the **STWHILEAG** rule, resulting in two proof obligations depicted in Figure 4.24. Using **ticl** we are able to reduce coinductive proofs to inductive premises, as is the case here. The right premise is a specification on the *loop body* of **rotate**. Proving it is straightforward,

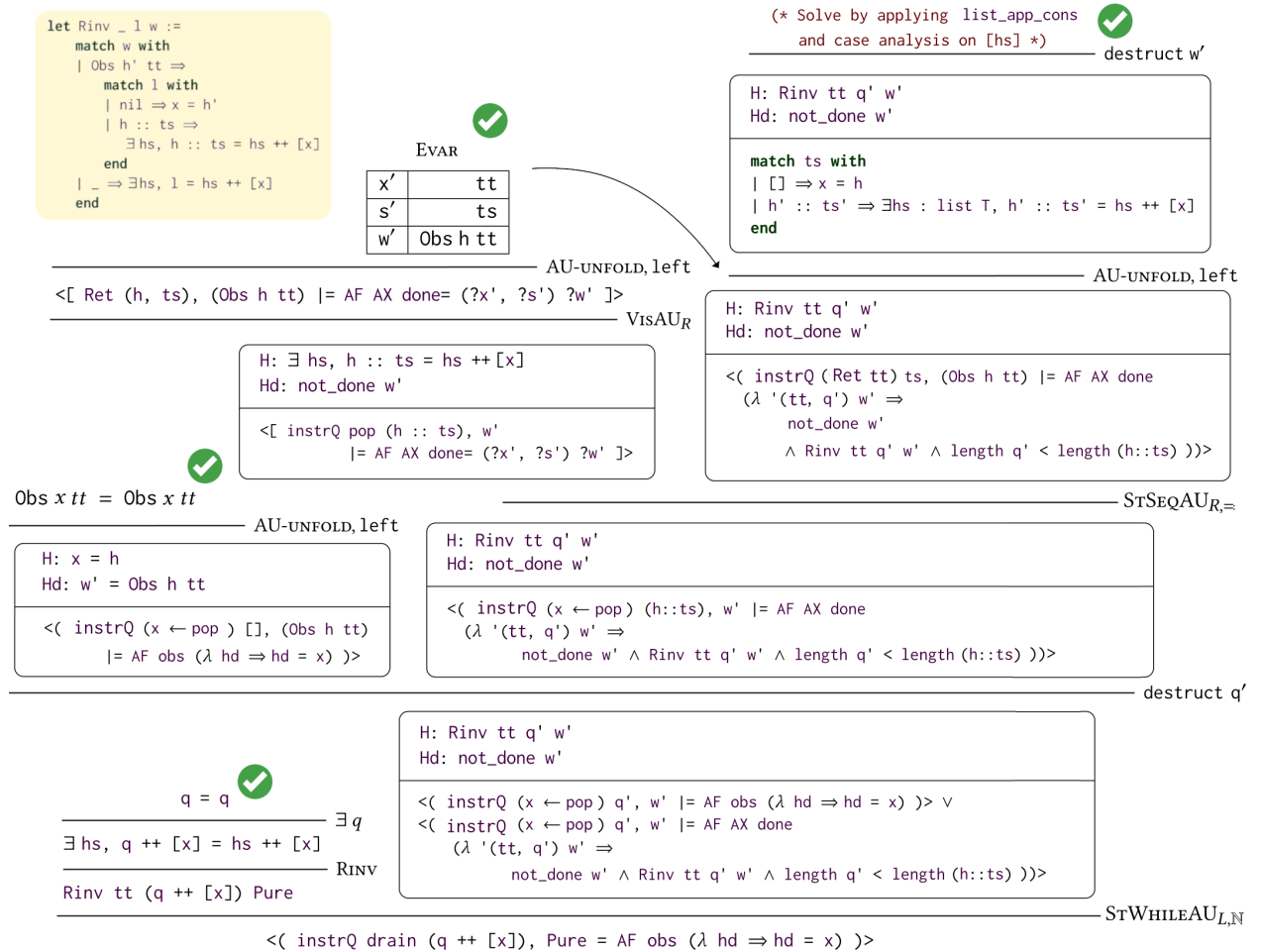


Figure 4.23: Proof that **drain** eventually observes x in the head position of the queue. The goal is in the bottom, work upwards by applying **ticl** structural lemmas and basic Coq tactics. Loop invariant **Rinv** is in the upper-left corner and loop variant is queue **length**.

we proceed by two applications of the sequence rule $\text{STSEQAU}_{R=}$ to get the value v popped, then to get the unit return value of `push v`. At that point, the loop body returns and the postcondition is satisfied.

The left premise $\langle (\text{instrQ rotate } q, w \models \text{AF obs } (\lambda hd \Rightarrow hd = x)) \rangle$ encodes the “eventually” part of “always-eventually”. We proceed by case analysis on the loop invariant:

1. If $h = x$, running the loop once will `pop` the target element x from the head and observe it, proving the property $(hd = x)$.
2. If $\exists i, \text{find } x \text{ ts} = \text{Some } i$, the target is in the tail of the queue `ts`. We proceed by applying the inductive lemma STWHILEAU_L with the loop invariant $\exists i, \text{find } x \text{ q} = \text{Some } i$, as the target will definitely be in the queue by the loop invariant, the ranking function `find x` will find the index of the target x . This index will get smaller every time, as it rotates closer to the head position.

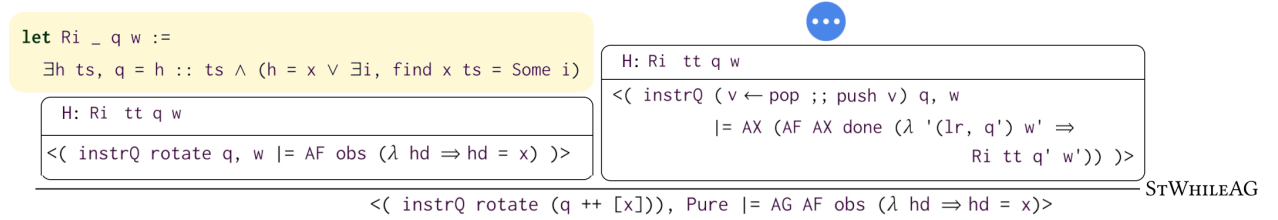


Figure 4.24: Beginning of “always-eventually” proof for `rotate`. Applying STITERAG using loop invariant Ri , leaves two finite proof obligations which are easy to conclude.

4.5.2. Secure Memory

For our next example we use the language $\text{StImp}_{\mathcal{S}}$, featuring a new heap $(\mathcal{M}_{\mathcal{S}})$, where every memory cell is tagged with a *label* (\mathcal{S}) . There are two security labels: *low* security (L) and high security (H). They form a preorder with respect to binary relation $l \leq l'$ — the smallest reflexive, transitive relation such that $L \leq H$ holds. Labelled memory is accessed by labelled instructions (`Read l_i x` and `Write l_i x y`). This scheme is inspired by Mandatory Access Control (MAC) systems, our goal is to detect secrecy violations — a low-security instruction should never access high-security memory.

The labelled memory semantics are given in terms of an instrumentation handler $h_{\mathcal{S}}$. We preserve

the heap semantics of the unlabelled heap $h_{\text{state}_{\mathcal{M}}}$. As we need to potentially access unlabelled heap variables during evaluation of labelled instructions (**Read** l_i ($x \in \text{AExp}_S$)), both the heap (\mathcal{M}) and labelled memory (\mathcal{M}_S) must be available to the instrumentation handler (h_S). We additionally instrument **StImp**_S reads (**Read** l_i x) over a memory cell $\llbracket x \rrbracket_{A,m} \hookrightarrow (l, n)$ with the pair of labels (l, l_i) , where l is the *memory cell label* and l_i is the *instruction label*. This way, if we observe a pair (l, l_i) such that $l_i \leq l$, this is indicative of a secrecy violation. Note we could easily enforce secrecy *dynamically*, by instrumenting **read** calls with a runtime check $l_i \leq l$ and forcing a deadlock on access violation. However, dynamic checks can hinder performance at runtime. By proving the *static* property **sec_safety_ag** we ensure safety without sacrificing runtime performance.

The two programs **sec_alice** and **sec_bob** in Figure 4.26 simulate two users: *Alice* who has high-security access, and *Bob* who has low-security access. Alice possesses a **secret** value which she writes on *odd* numbered addresses. Bob, on the other hand, will read from *even* numbered addresses. The nonterminating scheduler iterates over all the natural numbers and nondeterministically chooses either **sec_alice** or **sec_bob** to run each time. Our goal is to show that no secrecy violations occur.

The proof (Figure. 4.27) starts with the coinductive lemma **STWHILEAG**, which requires a loop invariant **Rinv'** and produces two proof obligations.

$$\begin{aligned}
S \in \text{Type} &= \quad | L \quad | H \\
\mathcal{M}_S \in \text{Type} &= \text{Map}_{\mathbb{N}, (\mathbb{N} * S)} \\
\text{AExp}_S \in \text{Type} &= \quad | \text{var } (s \in \text{string}) \quad | \dots \quad | \text{read}_{(l \in S)} (x \in \text{AExp}_S) \\
\text{BExp}_S \in \text{Type} &= \quad | (x \in \text{AExp}_Q) = (y \in \text{AExp}_Q) \quad | \dots \quad | \text{is_even } (x \in \text{AExp}_S) \\
\text{StImp}_S \in \text{Type} &= \quad | (s \in \text{string}) \leftarrow (y \in \text{AExp}_Q) \quad | \dots \quad | \text{write}_{(l \in S)} (x \in \text{AExp}_S) (y \in \text{AExp}_S) \\
h_S \in \text{state}_{\mathcal{M}} + E_S &\rightsquigarrow \text{InstrM}_{(\mathcal{M} * \mathcal{M}), (S * S)} \\
h_S (\text{Read } l_i \ x \in E_S) (m \in \mathcal{M}, \mu \in \mathcal{M}_S) &= \\
&\quad \text{let } (l, v) := \mu[\llbracket x \rrbracket_{A,m}] \text{ in Vis (Log } (l, l_i)) (\lambda _ \in \text{unit}) \Rightarrow \text{Ret } (v, (m, \mu)) \\
h_S (\text{Write } l_i \ x \ y \in E_S) (m \in \mathcal{M}, \mu \in \mathcal{M}_S) &= \text{Ret } ((), (m, (\llbracket x \rrbracket_{A,m} \hookrightarrow (l, \llbracket y \rrbracket_{A,m})) \cup \mu)) \\
h_S (e \in \text{state}_{\mathcal{M}}) (m \in \mathcal{M}, \mu \in \mathcal{M}_S) &= (h_{\text{state}_{\mathcal{M}}} e \ m) \gg \! \! \gg (\lambda '(v, m') \Rightarrow \text{Ret } (v, (m', \mu)))
\end{aligned}$$

Figure 4.25: Language **StImp**_S extends the language **StImp** with a global *labelled* memory where every address (\mathbb{N}) is tagged with either a *high* security (H) or *low* security (L) label.

<pre> Variable (secret: nat). Definition sec_alice := if is_even i then write H (i + 1) secret else write H i secret. </pre>	<pre> Definition sec_bob := if is_even i then read L i else read L (i + 1). </pre>	<pre> Definition sec_scheduler := i ← 0; do (sec_alice ⊕ sec_bob; i ← i + 1) while (true) </pre>
--	--	--

Figure 4.26: Alice (*High* security) writes *secret* to odd addresses and Bob (*Low* security) reads from even addresses, take their concurrent interleaving.

1. The *loop* satisfies the safety property $\mathbf{al} \leq \mathbf{ml} \text{ now}$, where \mathbf{al} is the instruction label and \mathbf{ml} is the memory label.
2. The *loop body* steps (outer \mathbf{AX}) then satisfies $\mathbf{al} \leq \mathbf{ml}$ until it terminates, at which point loop invariant \mathbf{Rinv}' is satisfied.

Lemma STWHILEAG hides the internal coinductive proof using the up-to principles in Figure 4.13, these technical details are never exposed to the user, who may use the lemma with little to no familiarity with the coinduction library [Pou16] and up-to principles.

The rest of the proof is straightforward. Proceed by examining both cases of the nondeterministic choice $(\mathbf{sec_alice}) \oplus (\mathbf{sec_bob})$ (due to the universal quantifier in \mathbf{AX} and \mathbf{AU}) using rule STBRAU_R. Then proceed by case analysis on whether \mathbf{i} is odd or even. We stop illustrating the proof in Figure 4.27 at this point in the interest of space. The four remaining subgoals are proved by observing the instrumentation of read/writes to labelled memory, then by simple reasoning about finite maps. The complete proof can be found in the Coq development.

4.5.3. Distributed Consensus

Our last example of a structural liveness proof is a distributed leader election protocol, running in a *unidirectional ring* configuration with three processes (Figure 4.30). Every process can only receive messages from the process on their right and send messages to the process on their left. The goal of the leader election protocol is to reach consensus across all processes and agree on a process to be the *leader*. Leader election is a common part of many distributed protocols, like Paxos [Lam01]. For simplicity we assume no network failures, no process failures, and no byzantine failures can occur. Modeling failures by using `ictree` nondeterminism is entirely possible, but doing so is beyond the

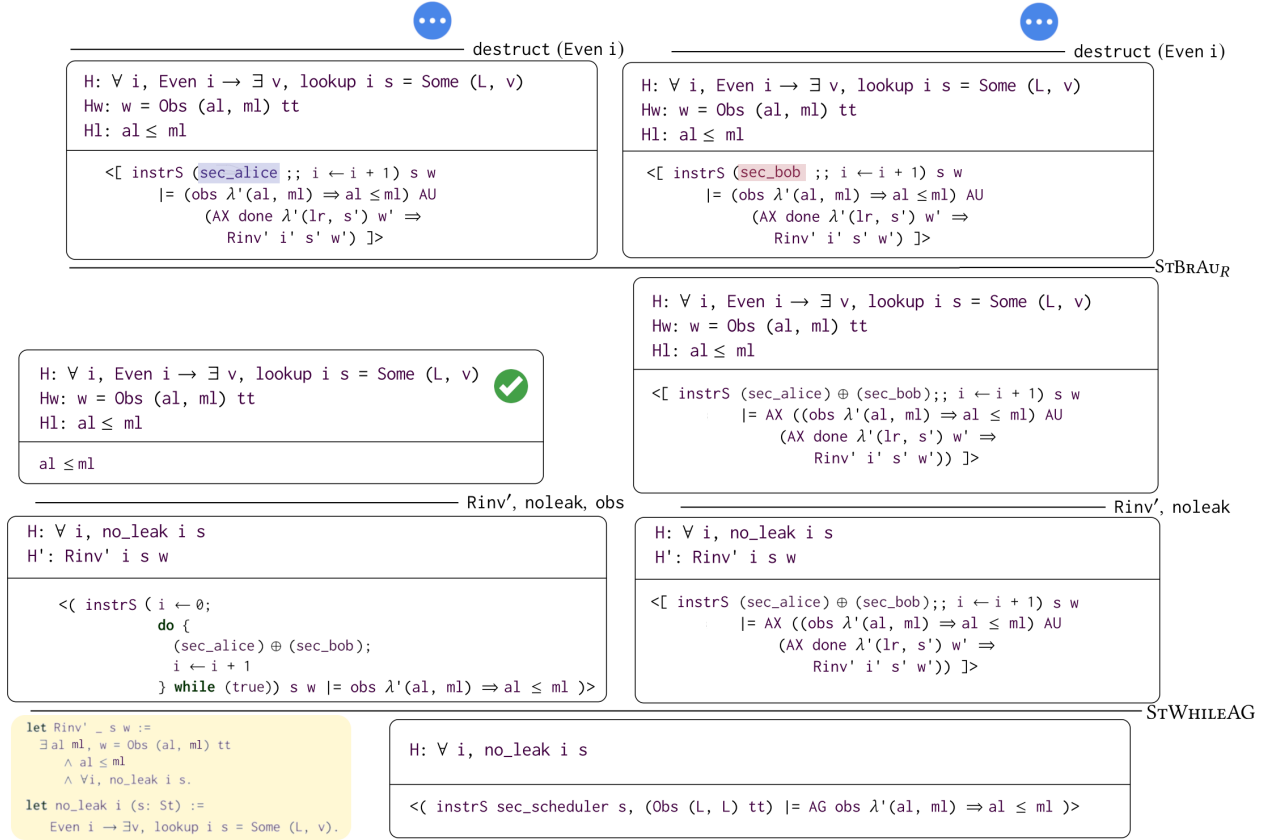


Figure 4.27: Beginning of concurrent shared memory proof. Discharging the scheduler and **AG** leaves as proof obligations finite proofs about Alice and Bob.

scope of this paper. The protocol is illustrated in Figure 4.30 and proceeds in two phases:

1. Proposing candidates.

- Initially every process self-nominates to be the leader (C_1, C_2, C_3).
- If the candidate ID received is greater than the process' own **pid** (this is only true for C_3), the message is propagated. Otherwise, the message is dropped.

2. Announcing the leader

- When a process receives their own candidacy message back, they announce they are now the elected leader (process 3 sends E_3).
- A process that receives message E_i (here $i = 3$) agrees PID i is now the leader and propagates the message.
- The last step continues forever and the protocol is nonterminating.

The programming language for processes requires several features orthogonal to the liveness of the protocol, such as sum types for the messages and pattern matching. We opt to use a shallow-embedding of $\text{ictree}_{E_{\text{net}}, X}$ in the Coq proof assistant which allows access to the entirety of Coq's programming language features. In addition, the *round-robbin* scheduler for a unidirectional ring requires stateful access to the *current process id*, we extend the scheduler with state events ($\text{state}_{\text{PID}_n}$).

We define messages for the election protocol (Msg_n) and assume basic vector random access operations on mailboxeds ($[\text{Msg}_n]_n$) in Figure 4.28. Then define message passing events (E_{net}) in Figure 4.29 for processes (proc) and scheduler state events ($\text{state}_{\text{PID}_n}$) for the round-robbin scheduler (rr) in Figure 4.31.

```

PIDn ∈ Type = fin' n
Msgn ∈ Type = | C (p ∈ PIDn) | E (p ∈ PIDn)
[Msgn]n ∈ Type = Vector n Msgn
(ms ∈ [Msgn]n)[p ∈ PIDn] ∈ Msgn [get message at index]
(ms ∈ [Msgn]n)[p ∈ PIDn] := (m ∈ Msgn) ∈ [Msgn]n [update mailbox at index]

```

Figure 4.28: Process identifiers (PID_n) and messages (Msg_n) are indexed by $n \in \mathbb{N}$, the number of processes in the protocol. The same is true for the mailboxes ($[\text{Msg}_n]_n$) — a vector of n messages, one for each process. Random access operations for vectors are assumed from Coq's standard library.

```

Enet ∈ Type → Type = | Send(m ∈ Msgn) | Recv
hnet ∈ Enet + statePIDn ~> InstrM(PIDn*[Msgn]n),(PIDn*Msgn)
```

$$\begin{aligned}
h_{\text{net}} (\text{Send } m \in E_{\text{net}}) (p \in \text{PID}_n, ms \in [\text{Msg}_n]_n) &= \\
&\text{Vis } (\text{Log } (p, m)) (\lambda _ \in \text{unit} \Rightarrow \text{Ret } ((), (p, ms[p + 1 \% n] := m))) \\
h_{\text{net}} (\text{Recv} \in E_{\text{net}}) (p \in \text{PID}_n, ms \in [\text{Msg}_n]_n) &= \\
&\text{Vis } (\text{Log } (p, m)) \lambda _ \in \text{unit} \Rightarrow \text{Ret } (ms[p], (p, ms)) \\
h_{\text{net}} (\text{Get} \in \text{state}_{\text{PID}_n, \text{PID}_n}) (p \in \text{PID}_n, ms \in [\text{Msg}_n]_n) &= \text{Ret } (p, (p, ms)) \\
h_{\text{net}} (\text{Put } p' \in \text{state}_{\text{PID}_n, \text{unit}}) (_ \in \text{PID}_n, ms \in [\text{Msg}_n]_n) &= \text{Ret } (p', (p', ms))
\end{aligned}$$

Figure 4.29: Send and receive events (E_{net}) performed by each process in the leader election protocol. Get and put events ($\text{state}_{\text{PID}_n}$) performed by the round-robbin scheduler to track the current running process (PID_n).

The leader election protocol starts with candidate messages (*Phase 1*) already in the mailboxes of their respective processes. This is visible in the specification `election_live` in Figure 4.31 and the initial $[\text{Msg}_n]_n$ state is `[C 3;C 1;C 2]`. The target property for this protocol is the liveness property

“eventually the highest PID will be elected the leader”. We start by taking cases on the initial nondeterministic choice of `rr`, depending on which process starts

1. $PID = 1$: Process 1 receives the candidacy message `C 3` and propagates it, process 2 propagates it as well, process 3 receives their own candidacy (`C 3`) and switches to *Phase 2*.
2. $PID = 2$: Process 2 receives the candidacy message `C 1` and drops it, process 3 receives the candidacy `C 2` and drops it, as it less than their own PID. Process 1 receives the candidacy message `C 3` and propagates it, the proof is the same as $PID = 1$ at this point.
3. $PID = 3$ process 3 receives the candidacy `C 2` and drops it, as it less than their own PID. Process 1 receives the candidacy message `C 3` and propagates it, the proof is the same as $PID = 1$ at this point.

Similarly we proceed in *Phase 2*, where process 1 received the *elected* message (`E 3`) of process 3 and propagates it to process 2, who propagates it to process 3. At this point, the *eventually* property is satisfied, the current process 3 has received their own elected message (`E 3`) and we conclude the proof. The proof proceeds by stepping the system a finite number of times, as shown in Figure 4.30 and using the $BINDAU_{R=}$ lemma to interpret each call to `proc cid`.

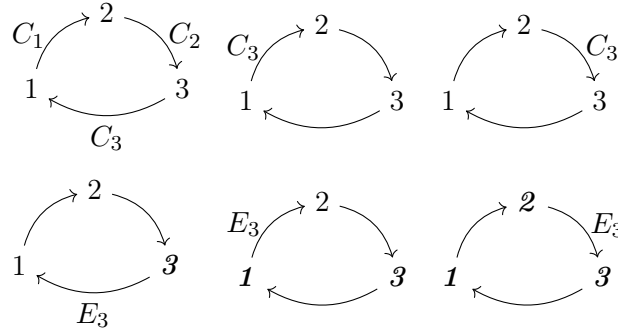


Figure 4.30: A unidirectional ring with three processes running the leader election protocol.

4.6. Discussion and Related Work

There are numerous, thoroughly studied model checking systems [BCI⁺16, Hol97, YML99] which are used today for computer systems verification [SMG⁺24, HHK⁺15, Hol97, Lam94], We do not intend to compete with established model checking platforms and position `ticl` with structural

program logics.

Iris and Transfinite Iris: Iris [JKJ⁺18] is a concurrent-separation logic framework for Coq that uses step-indexed logical relations to prove safety properties of concurrent programs. The recent extension *Transfinite Iris* [SGG⁺21] 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 $\models_{L,R}$ in **ticl**. At the same time, **ticl**, unlike Iris, has no facilities for 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. [LCK⁺23] 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**.

```

(* Election protocol participant *)
Definition proc (pid: Pid) :=
  m ← recv ;;
  match m with
  | C candidate =>
    match compare candidate pid with
    (* Propagate [candidate] *)
    | Gt => send (C candidate)
    (* Drop [candidate] message *)
    | Lt => Ret tt
    (* [pid] was elected, send [E pid] *)
    | Eq => send (E pid)
    end
  | E leader => send (E leader)
end.

(* Infinite round-robin scheduler *)
Definition rr :=
  (* Nondeterministic first pick *)
  cid ← branch n ;;
  iter (λ _ =>
    proc cid ;;
    cid ← (cid + 1) % n ;;
    Ret (inl tt)
  ) tt

(* Leader election liveness *)
Definition election_live :=
  <( instr h_net rr (F1, [C 3; C 1; C 2]), Pure
    |= AF obs (λ '(cid, msg) =>
      cid = 3 ∧ msg = E 3) )>.

```

Figure 4.31: Process **proc** and a round-robin scheduler **rr** implement the leader election protocol. The goal specification is: process $\text{cid} = 3$ *eventually* receive their own *elected* message (E 3) back.

Maude: The Maude language and Temporal Rewriting Logic (TLR) [Mes08, Mes92] recognize the benefits of structural approaches (namely term rewriting) to temporal logic verification. In `ticl` we enable term rewriting with *up-to-guard equivalence* under a temporal context (Section 4.3.1). However, Maude operates on the level of models, not on the level of executable programs. This creates a gap between the executable code and the properties verified. In addition, deadlocked programs (\emptyset) are not supported which makes working with monadic programs difficult, as we explain in Subsection 4.3.2.

Dijkstra monads: Several works on Dijkstra monads target partial-correctness properties in the style of weakest preconditions [AHM⁺17, MAA⁺19, WAH⁺22, SZ21]. Recent work targets total-correctness properties like “always” [SZ21] but not general temporal properties like liveness.

CTL in Coq: Doczkal et al. [DS16] develop an embedding of CTL in Coq for the purpose of proving completeness and decidability over Kripke automata. Their automata are left-total; every world w has an \mathcal{R} successor, where \mathcal{R} is the transition relation. This precludes terminating programs and deadlocked programs. We give a different encoding in Section 4.3.2 that works with monads and we are able to prove monad and iterator lemmas for our models in Section 4.4.1.

Synthesising ranking functions: Yao et al. [YTGN24] 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 be used with `ticl` lemmas like `STWHILEAUL,ℕ` to get mostly automated, formal proofs of liveness.

4.6.1. Conclusion

In this work we ask: is it possible to write structural proofs in a general temporal logic akin to proofs in Hoare logic? We believe we have answered affirmatively, and in the process of answering the question we developed Temporal Interaction and Choice Logic (`ticl`), a specification language capable of expressing general liveness and safety properties (we summarize `ticl` in Figure 4.9). Along the way,

we also designed an extensive metatheory of syntax-directed lemmas (Figures 4.14, 4.15, 4.16, 4.20) that encapsulate complex (co-)inductive proofs to simple rule application and rewriting. We applied `ticl` to several examples from T2 CTL benchmark suite [BCI⁺16] and in four examples inspired from computer systems as a way to demonstrate the metatheory in action.

CHAPTER 5

Proposal: a Language and Compiler for Zero-knowledge Protocols (Zippel)

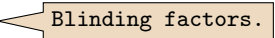
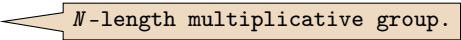
Over the last decade we have seen a surge of efficient, non-interactive, zero-knowledge (NIZK) protocols. While the basic building blocks of proof systems are by-and-large standardized, the same could not be further from the truth when it comes to implementations. Each proof system has its own ad-hoc implementation, using combinations of different open-source libraries, allowing for implementation bugs to appear. With regards to performance, implementations are hand-optimized by low-level tinkering with Rust’s thread primitives and do not compose easily, mostly due to lack of standardization. Existing libraries like Arkworks [ac22] fill the standardization gap, but lack support for automatic parallelization and are still fairly low-level. Finally, the proof that a protocol is zero-knowledge is done by pen-and-paper, disconnected from the implementation and often leading to knowledge leaks [CET⁺24].

Zippel is a programming environment for proof systems, aimed at improving the safety, performance and composition of cryptographic protocols. To motivate the design of Zippel we study several established proof systems; Schnor’s [Sch91a], KZG [KZG10], Groth16 [Gro16], Spartan [Set20], Marlin [CHM⁺20], Hyrax [WTS⁺18], Bulletproofs [BBB⁺18], Nova [KST22b], Plonk [GWC19] and HyperPlonk [CBBZ23]. We identify their common characteristics informing our design of the Zippel programming and specification language. First, we look at a proof system’s on-paper mathematical description, briefly explaining the notation, cryptographic assumptions and define the notions of soundness, completeness and zero-knowledge. Then we identify mathematical objects that appear in these proof systems and design a type system around these objects. We identify operations, like arithmetic, vector operations, polynomial operations, lagrange interpolation, multi-scalar multiplication, recursion, transcripts etc that proof systems routinely use and Zippel must support.

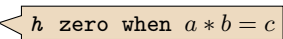
Finally, we propose a static analysis based on information flow security types, that efficiently proves no unintended information is leaked between protocol participants. The intuition behind proving zero-knowledge mechanically lies in simulator arguments, which can be reduced to an argument about

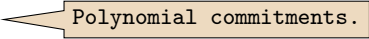
```

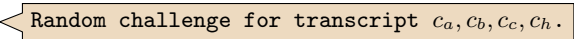
/// Prove (P) to verifier (V) in ZK that [a * b == c]
proto hadamard<N, F, P, V>(a b c: private<P> [F; N]) where a * b == c {

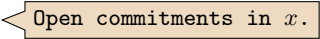
  let [r_a, r_b, r_c] = rand<[F;3]>; 
  let d = gen<N>; 

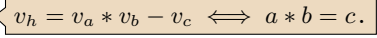
  let z_a = interpolate(a, d) + r_a * Poly::vp(d);
  let z_b = interpolate(b, d) + r_b * Poly::vp(d);
  let z_c = interpolate(c, d) + r_c * Poly::vp(d);

  let h = (z_a * z_b - z_c) / vp(d); 

  [c_a, c_b, c_c, c_h] <- [
    PC::commit(z_a), 
    PC::commit(z_b),
    PC::commit(z_c),
    PC::commit(h)
  ];

  x <- challenge<F, V>; 

  [pi_a, v_a] <- PC::open(c_a, x);
  [pi_b, v_b] <- PC::open(c_b, x);
  [pi_c, v_c] <- PC::open(c_c, x);
  [pi_h, v_h] <- PC::open(c_h, x); 

  assert<V>(v_h == v_a * v_b - v_c 
    && PC::verify(pi_a, c_a, x, v_a)
    && PC::verify(pi_b, c_b, x, v_b)
    && PC::verify(pi_c, c_c, x, v_c),
    && PC::verify(pi_h, c_h, x, v_h));

```

Figure 5.1: *Zippel* protocol between P , V proves the *Hadamard* product $a*b = c$ in Zero-Knowledge.

indistinguishability of finite traces, sampled from uniform random distributions.

5.1. Example 1: Hadamard product

We jump into our first Zippel protocol `hadamard` in Figure 5.1 implementing a zero-knowledge protocol for a prover (\mathcal{P}) demonstrating to a verifier (\mathcal{V}) knowledge of a vector (c) that corresponds to the Hadamard (pairwise) product of two vectors ($a * b == c$).

Despite this being a very simple example, the protocol is concise, clear, and maps closely to the mathematical description of the protocol. Second, Zippel expresses both the prover and the verifier in the same Zippel protocol implementation by annotating the verifier's work as an assertion with type " \mathcal{V} ". It then outputs separate Prover and Verifier modules that developers can incorporate

into larger applications. Third, Zippel deals with all aspects of parallelism (thread allocation, scheduling, etc.) automatically while the developer needs to do this manually and in an ad-hoc way. By tracking explicit (and implicit) data dependencies we can generate a dataflow graph seen in Figure 5.2 and color each independent section with a different color. Each section will be run by a different thread pool, according to their associated costs. Last, if a developer forgets to use a blinding factor (Line 4 in Zippel) for a variable with a secret type, Zippel automatically detects this and informs the developer that the verifier would be able to learn information about a, b, c thus violating zero-knowledge.

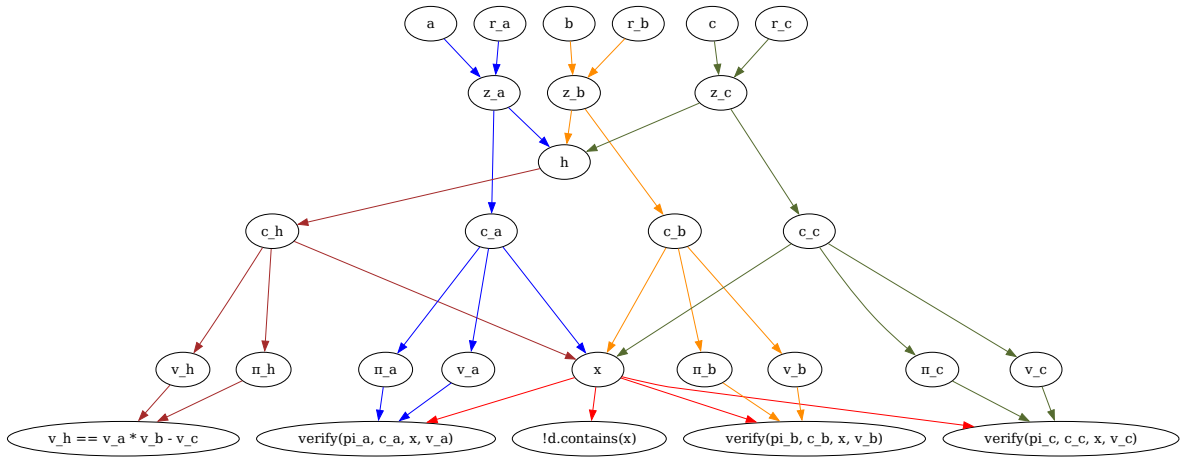


Figure 5.2: Dataflow graph of `hadamard` protocol.

In the hadamard example we incorporate polynomial commitments, note those are not primitives in Zippel but rather standard library calls abstracted in Zippel using functions. In the next example we formally prove zero-knowledge and show how those arguments works in Zippel.

5.2. Schnorr's ZK protocol

Schnorr's protocol for user identification [Sch91a] involves a Prover (\mathcal{P}) convincing a Verifier (\mathcal{V}) that he knows a private key (s), such that $g^s \bmod p \equiv x$. The security of the protocol depends on the discrete logarithm assumption. Figure 5.3 shows the protocol diagram for Schnorr's, the protocol works because if $z = r + c \cdot s$ then $g^z \bmod p \equiv g^{r+c \cdot s} \bmod p \equiv g^r \cdot g^{c \cdot s} \bmod p \equiv g^r \cdot (g^s)^c \bmod p \equiv t \cdot x^c \bmod p$.

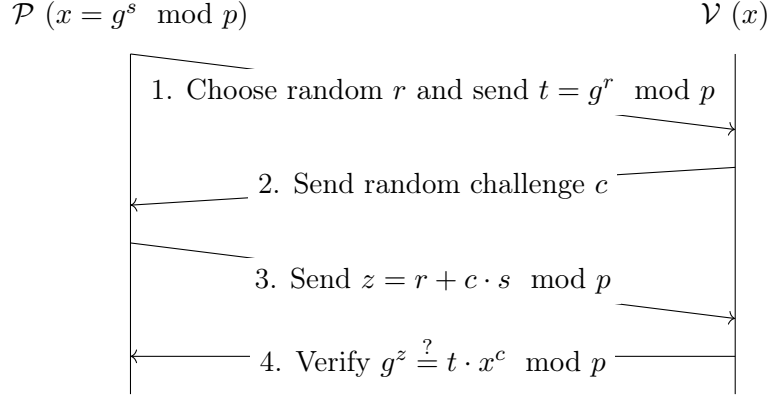


Figure 5.3: Schnorr's protocol for zero-knowledge identification.

Schnorr's protocol satisfies a property called *Honest Verifier Zero-Knowledge* (HVZK), meaning if the verifier (\mathcal{V}) samples c randomly and not adaptively, then they can gain no knowledge from the transcript (messages (t, c, z)).

Proof: Proofs of zero-knowledge typically involve a *simulator* argument. A simulator is a process that simulates the prover \mathcal{P} without access to the secret s . If the simulator can convince the (honest) verifier, while the simulator transcript and the real prover transcript are *computationally indistinguishable*, then the protocol is zero-knowledge. For Schnorr's in Figure 5.3 the simulator is simply running the protocol in reverse, by sampling z and computing t

$$S(x) = \{z \xrightarrow{\$} \mathbb{Z}_p; c \xrightarrow{\$} \mathbb{Z}_p; t \leftarrow \frac{g^z}{x^c}\}$$

The verification condition is satisfied for $S(x)$ as $\frac{g^z}{x^c} \cdot x^c = g^z$ and the real trace and the simulator trace are indistinguishable from the verifier's perspective

$$(g^r, c \xrightarrow{\$} \mathbb{Z}_p, r + c \cdot s) \approx (\frac{g^z}{x^c}, c \xrightarrow{\$} \mathbb{Z}_p, z \xrightarrow{\$} \mathbb{Z}_p)$$

This proof does not show perfect zero-knowledge, only HVZK, because a dishonest verifier \mathcal{V} could choose c adaptively, making the two transcripts no longer indistinguishable. The focus of this proposal is in NIZK protocols in which the verifier is replaced with random oracle calls — through

a process called the Fiat-Shamir transformation [FS86]. For NIZKs the HVZK model is sufficiently secure.

From the perspective of proof mechanization, simulator arguments are hard to automate as they would require automated synthesis of the simulator process ($S(x)$). From a machine-checked proof perspective, if given the simulator for Schnorr’s $S(x)$ we can easily see the two transcripts are indistinguishable – r, c are sampled through a uniform random distribution, so $r + c \cdot s \approx z \xrightarrow{\$} \mathbb{Z}_p$ and $g^r \approx \frac{g^z}{x^c}$ are both indistinguishable from random. We posit it is possible to “approximate” simulator proofs by performing a taint analysis on secret inputs (here s) and showing they are always “blinded” by uniform random elements (c). Our goal is to build and verify proof objects approximating simulator proofs of zero-knowledge, using machine-checked verification techniques.

5.2.1. Example 2: Schnorr

Figure 5.4 shows another Zippel protocol, Schnorr’s protocol (Figure 5.3). Note the the *relation clause* `where gen<G>^s = x` specifying the relation to be proved by the protocol. Schnorr’s protocol is sound if after its completion the verifier \mathcal{V} is convinced of the validity of relation $\text{gen}\langle G \rangle^s = x$ for a value x .

```

/// Prove (P) to verifier (V) in ZK that [g^s = x]
proto schnorr<V, P, G: Group, F: Scalar<G>>(private<P> s: F, public x : G) where
  gen<G>^s == x {
    t <- gen<G>^s;
    public c <- challenge <V, F>;
    z <- r + c * s;
    assert<V>(gen<G>^z == t * x^c)
  }

```

Figure 5.4: Schnorr’s protocol in *Zippel* run by \mathcal{P} via the Fiat-Shamir transformation.

The goal is to implement an SMT-based verification procedure that checks *soundness* and *completeness* of the protocol, finding counter-examples if they exist. In this case, a completeness counter example would be values s, x that satisfy the relation $\text{gen}\langle G \rangle^s = x$ but for which the final verifier assertion `assert<V>(gen<G>^z == t * x^c)` does not hold — fortunately such values do exist.

We have not yet confirmed SMT-proofs are applicable to *computational soundness*, which is the

golden standard for cryptographic protocol verification. We posit the Computationally Complete Symbolic Attacker (CCSA) model [BCL12] can offer solutions by axiomatizing computational soundness to symbolic soundness for the purposes of automatic verification. At the moment there is no plan to reason probabilistically using verification tools such as EasyCrypt [BDG⁺12]. The learning-curve of these tools is rather steep and our primary goal is usability by experts in cryptography, but not necessarily verification.

With regards to types of variables that appear in the `schnorr` protocol (Figure 5.4) there are different qualifiers attached

- `public` is a *principal* qualifier indicating `x` is known to both prover \mathcal{P} and verifier \mathcal{V} .
- `private<P>` is a principal qualifier indicating `s` is *known* only to \mathcal{P}
- Type `G`: `Group` indicates `x` belongs in the group of the underlying elliptic curve. Instantiating `G` to a specific curve is delayed until after compilation, so `G` is treated abstractly.
- Type `F`, where `F`: `Scalar<G>` indicating the associated scalar field of the abstract group `G`. Scalar multiplication (`g~s` in multiplicative notation) is only allowed between groups and their scalar fields.

In the following section we go into more depth on datatypes in Zippel, including groups, fields, vectors, matrices, univariate and multilinear polynomials.

5.2.2. Fiat-shamir challenges

If we were to use Schnorr’s protocol (Figure 5.3) in practice, both \mathcal{P} and \mathcal{V} would need to interact with each other a few times over the wide area network, creating performance and synchronization issues. Instead, we can use the Fiat-Shamir transformation to turn an interactive protocol into an NIZK [FS86]. The Fiat-Shamir transformation allows a prover \mathcal{P} to *simulate* to simulate an honest verifier \mathcal{V} , using calls to a random oracle. In Zippel random oracle calls are implemented through the `challenge<V, F>` command.

However, there is a potential pitfall — a security vulnerability called “The Frozen heart” vulnerability [DMWG23]. Continuing with Schnorr’s protocol, if `c` is picked without taking into account `t` a

dishonest prover \mathcal{P} could convince the verifier even without knowing the private key s . This causes the introduction of implicit data dependencies in Fiat-Shamir protocols, even when no data dependency exists in the protocol. We make Fiat-Shamir data dependencies explicit in Zippel, through the use of *Transcript bindings* ($t \leftarrow \text{gen}\langle G \rangle \sim s$) used in Figure 5.4. Transcript bindings are variation on let-bindings that assign the value $\text{gen}\langle G \rangle \sim s$ to variable t , but also appends it to a global, shared, append-only data-structure we call the *transcript*. When a random oracle `challenge` is invoked, Zippel will hash the current transcript by calling $\text{Hash}(t \oplus h)$, where h is a random seed. More details about use of *transcripts* are in (Section 5.3.2), including a comparison to the SAFE API [AKMQ23] and how Zippel simplifies Fiat-Shamir safety without the need for explicit annotations.

5.3. Zippel language

The zippel language is a pure, strictly-typed imperative language, featuring a combination of sized-types and information-flow type system, tracking the *sizes* (sizes s) of data and their *owners* (principals P). We start with the type system and proceed to the core language grammar, both can be found in Figure 5.5.

5.3.1. Zippel type system

Zippel’s base types simulate the algebraic structures commonly used in Proof systems; elliptic curve groups, fields and pairing friendly groups. Those structures have strict rules about which operations are allowed and which disallowed, for example, group elements can not be multiplied, only added and subtracted, group scalars can multiply their associated group elements but not others etc.

Additionally, Zippel features sized container types; vectors, matrices, univariate and multilinear polynomials, defined in Figure 5.5 and used in the `hadamard` example in Figure 5.1. Zippel’s sized types ensure common mistakes such as adding vectors of different sizes, interpolation errors, polynomial arithmetic errors and more, get caught at compile-time.

The sized type system depends on distinguishing between *exponent-of-two* sizes and *linear* sizes, we will see why exponents of two are important when we introduce the concept of logarithmic recursion. The intuition is, we would like to be able to unfold logarithmic recursion over vectors or

polynomials of degree 2^N exactly N times, before scheduling.

Elliptic curve groups

Elliptic curves are smooth, projective algebraic curves defined by an equation of the form:

$$y^2 = x^3 + ax + b$$

where a and b are constants. Elliptic curves are widely used in cryptography due to their rich algebraic structure, allowing for operations such as point addition and scalar multiplication. These operations form a group (\mathbb{G}), making elliptic curves ideal for constructing secure cryptographic schemes. In our case studies we found protocols abstract over the specific curve and operate on the level of abstract algebra (groups, fields etc). Zippel allows programming over groups and fields abstractly, using generic types like \mathbf{G} in Figure 5.4.

Finite field scalars

Protocols often operate on scalar, finite-field elements $f \in \mathbb{F}_q$ modulo a large prime number q . Finite field elements are operated on by field arithmetic operations; addition (+), subtraction (−), multiplication (*), division (/), exponentiation (^), modulo a prime q . The usual algebraic rules of field arithmetic apply; closure, associativity, commutativity, distributivity, symmetry, unit and inverses for addition and multiplication.

Efficient implementations of finite field arithmetic exist in Arkworks [ac22] and we inherit their benefits in Zippel with two field “kinds”; scalar finite field of an associated group, (2) generic finite field elements with no associated group. We use the kind **F**: **Scalar**< \mathbf{G} > to associate a scalar field with its group and use *kind inference* when possible, to infer these type constraints without user annotations.

Pairing-friendly curves

Protocols often operate over a pair of elliptic curves **Pairing**($\mathbb{G}_1, \mathbb{G}_2$), where \mathbb{G}_1 refers to the algebraic group induced by the first elliptic curve, and \mathbb{G}_2 to the second curve. The key property of pairing-friendly curves is a bilinear map operation we call “multiplication”, such that

$g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, $g_1 * g_2 \in \mathbb{G}_T$, where \mathbb{G}_T is a third *target* curve. Pairing-friendly curves allow for concise, constant-size proofs and fast verification times, making them well-suited for the succinct and non-interactive properties required in SNARKs [Gro16]. For example pairing-friendly BLS12-381 [Bow17] has been successfully used in proof systems for Zcash.

Efficient implementations of common pairing-friendly elliptic curves exist in Arkworks [ac22] and we incorporate those into Zippel. Designate a pairing friendly curve pair using kind `G: Pair(G1, G2)` to indicate `G` is the projective group of pairing `G1`, `G2`.

Univariate polynomials

Univariate polynomials ($P(X \in \mathbb{F}_q)$) over finite fields are common in proof systems. Many efficient polynomial commitment schemes target univariate polynomials [KZG10]. Informally, polynomial commitment schemes allow a prover to produce a commitment c to a univariate polynomial $P(X)$, and later “open” $P(X)$ at any point ($z \in \mathbb{F}_q$), producing an evaluation proof π showing that the opened value is consistent with the polynomial commitment c at z . Univariate polynomials are the building blocks of Marlin [CHM⁺20] and Plonk [Gur21]. In Zippel univariate polynomials have their own sized-type (here size means degree of the polynomial) represented as `Uni<T, s>` where `T` is a field-type and `s` is a size.

Multilinear polynomials

A multilinear polynomial $M(X_0 \in \mathbb{F}_q, \dots, X_n \in \mathbb{F}_q)$ over multiple variables must be linear (degree at least 1) on every variable. For example, $M(x, y) = 2x + xy + 3y + 4$ is linear on both x, y . As in univariate polynomials, multilinear polynomials are defined over finite-fields (\mathbb{F}_q). Efficient commitment schemes for multilinear polynomials exist and are the building blocks of Spartan [Set20] and Hyperplonk [CBBZ23].

Vectors

Vectors of size n where elements are field elements ($\vec{v} \in \mathbb{F}_q^n$) or group elements ($\vec{v} \in \mathbb{G}^n$) appear often in proof systems. Efficient commitment schemes exist to verify the inner-product of vectors called *Inner Product Arguments*(IPA), for example bulletproofs [BBB⁺18] and Hyrax [WTS⁺18].

Several circuit languages like *rank 1 constraint systems* (R1CS) commonly used to express general-purpose computation as products of vectors and matrices – which we represent as nested vectors — can be efficiently verified with IPAs.

5.3.2. Operations

Now we identify high-level cryptographic operations that operate on the core datatypes. Many of these operations are parallelizable and fast, well-studied implementations exist. We implement them as first-class operations in Zippel, meaning the compiler and backend will work together to optimize their execution.

Lagrange interpolation

By combining the power of polynomials and vectors we can get more efficient and expressive protocols, that alternate between polynomial commitments and vector commitments. Changing polynomial representations from the *evaluation domain* (a vector of points) to the *lagrange domain* (a polynomial in coefficient form) requires support for polynomial interpolation, also called *Lagrange interpolation*. Several high-performance, multithreaded implementations of Lagrange interpolation exist, based on the inverse Fast-Fourier Transform (FFT) algorithm.

Multiscalar multiplication

Multiscalar multiplication is a cryptographic operation that computes the sum of multiple elliptic curve points, each scaled by a corresponding scalar. The following is a multiscalar multiplication

$$\sum_{i=1}^n f_i * g_i$$

Where $f_i \in \mathbb{F}_q$ and $g_i \in \mathbb{G}$. Pippenger’s algorithm optimizes multiscalar multiplication by grouping scalar bits and accumulating points into buckets, reducing the number of point additions and scalar multiplications. In a multithreaded implementation, Pippenger’s algorithm is parallelized by distributing the work of bucket accumulation, scalar preprocessing, and point addition across multiple threads. This approach significantly improves efficiency, especially for large inputs, making it ideal for cryptographic protocols that require fast, large-scale multiscalar multiplications.

Multiscalar multiplication appears in most proof systems defined over elliptic curve groups and it must be performed efficiently in Zippel. We give a special version of *inner product* that can deduce by the types of its operands when to use Pippenger’s algorithm, taking advantage of multithreading support of the host hardware.

Cryptographic transcript

NIZK protocols are commonly produced from interactive zero-knowledge protocols via the Fiat-Shamir transformation [FS86]. Instead of the *Verifier*(\mathcal{V}) sending the *Prover*(\mathcal{P}) a challenge, \mathcal{P} can sample a challenge from a random oracle to “simulate” challenges from \mathcal{V} . In practice, the random oracle is heuristically instantiated with a cryptographic collision-resistant hash function given as input the public interaction from \mathcal{P} up until that point ($\text{Hash}(\mathcal{P}_0 \oplus \dots \oplus \mathcal{P}_n \oplus h)$). Clever attacks on naive implementations of the Fiat-Shamir transformation are possible [BPW12, DMWG23], the key insight from those attacks is randomness cannot be crafted freely and has to be a function of the verifier’s past knowledge. Multiple implementations of cryptographic transcript libraries exist written in Rust, most notably the SAFE API [AKMQ23] requires access patterns to the transcript to be declared ahead of time. We can alleviate this requirement in Zippel, by taking advantage of the transcript dependencies introduced in Zippel programs.

Logarithmic Recursion

Recursive proofs enable the composition of multiple proof systems, allowing the verification of complex computations in a scalable and efficient manner. By recursively proving smaller components of a computation, the size and verification time of the final proof remain sublinear in size, regardless of the complexity or depth of the original computation. This is particularly beneficial for applications requiring repeated or incremental proofs, such as in blockchain rollups, where verifying large state transitions efficiently is critical. Notable examples include Bulletproofs [BBB⁺18], which reduce proof sizes without a trusted setup, and Nova [KST22b] which enables recursive SNARKs for general-purpose computations with significant efficiency improvements in proof generation and verification.

In Zippel we implement a restricted kind of recursion called *sized logarithmic recursion*. As Zippel does not have conditionals in its expression language (Figure 5.5) it is tricky to differentiate between

the base case and the recursive case. We use sized types to allow for recursion on the size of containers, with an added restriction; the recursive call should have at least one argument with size 2^n for some $n > 0$. An example of a recursive function adding up the elements of a vector is in Figure 5.5. Note, in the base case — which is omitted in Zippel — the size of input vector v is $N = 1$ which we simply dereference to return the inner element,

In the recursive case, split the vector in two slices $v[0..2^{N-1}]$ and $v[2^{N-1}..2^N]$, each of the slices has length 2^{N-1} . This satisfies the *logarithmic* step of recursion, which allows us to unroll the recursion $\log(N)$ times. In the base case, the vector size will be $2^0 = 1$ so we simply dereference the inner element. The Zippel sized type system will feature two different “kinds” of size types, exponents and linear sizes. While exponents can be used in logarithmic recursion, linear sizes can be concatenated, subtracted and multiplied by a constant, useful for polynomial multiplication, division and exponentiation respectively.

The zippel compiler will give an error on recursive calls which do not fulfill the logarithmic restriction (container has size 2^n for some n). This limited form of recursion allows us to implement recursive proof systems while having the ability to unroll recursion and optimize across recursive calls.

```
/// Sum of elements in a vector, recursive case
fn sum<F, N>(v: [F; 2^N]) -> F :=
  sum(v[0..2^N/2]) + sum(v[2^N/2..2^N])
```

Figure 5.5: Recursively sum elements of a vector in Zippel using *sized logarithmic recursion*.

User-defined operations

We purposefully limit the syntax of Zippel, to a small, well-typed set of commonly used operations, but we do not claim this is a complete representation of everything protocol developers need.

The zkSNARK ecosystem has been built on the Rust programming language offering the advantages of performance and memory safety while maintaining a high-level programming interface. In Zippel we plan to offer an “escape-hatch” for programmers to write their own custom Rust functions and use them inside Zippel programs as black boxes. At the same time, using Rust inside Zippel misses on the high-level optimizations we implement for zippel operations, so small, modular extensions

should be preferred over big Rust subroutines.

5.3.3. Zippel expression language

Now we proceed to define the expression language of Zippel in Figure 5.5. There are two classes of expressions, *arithmetic expressions* (exp) and boolean expressions ($beexp$). Arithmetic expressions are used to calculate polynomials, group elements, field elements, as well as map-comprehensions, reductions and function application. Boolean expressions are used for *assert* statements, checked by either the verifier \mathcal{V} or prover \mathcal{P} .

5.3.4. Zippel declaration language

In the top-level Zippel definitions are built as functions and protocols in Figure 5.5. While protocols describe interactions between principal parties, functions are simply used for abstraction and recursion. While protocols do not have a return value, functions do. The job of a protocol declaration is to make a convincing argument that its relation is satisfied, not to compute. Contrary, functions compute values of any datatype and can perform recursion.

5.4. Conclusions and future work

The Zippel programming language and compiler are works in progress, as such the grammar might change, while the core mission will not. Zippel is a programming environment for proof systems, aimed at improving the safety, performance and composition of cryptographic protocols. The design of Zippel was informed by a study of several established proof systems, including:

- Schnorr’s identification protocol [Sch91a].
- KZG polynomial commitments [KZG10].
- Groth16 proof system [Gro11].
- Spartan proof system [Set20].
- Marlin proof system [CHM⁺20].
- Hyrax vector commitments [WTS⁺18].
- Bulletproofs inner-product argument [BBB⁺18].
- Nova recursive proof system [KST22b].

- Plonk proof system [GWC19].

We plan to implement many of the protocol use cases and expect to see better performance and safety than the state of the art. Zippel aims to provide a higher level of abstraction and safety than existing libraries like Arkworks. The goal is to allow developers to write cryptographic protocols in a more concise and secure way, while also taking advantage of automatic parallelization and memory layout optimizations. Zippel will include features such as:

- Static safety features that prohibit common mistakes in protocol design.
- Automatic parallelization and memory layout optimizations.
- An information-flow principal type system that tracks the owners of data and catches zero-knowledge violations at compile-time.
- Support for high-level cryptographic operations such as Lagrange interpolation, multiscalar multiplication, and cryptographic transcripts.
- Extensibility through a Rust foreign-function interface.

5.5. Zippel language grammar

lin, i, j	$::=$	Positive linear arithmetic expression
	n	positive integer literal
	T	positive integer variable
	$lin + lin'$	positive linear addition
	$n * T$	positive linear product
pow, b	$::=$	Power of two
	$2 \wedge i$	linear positive exponent of two
	$b * b'$	multiply exponents of two
$size, s$	$::=$	Sizes as dyadic numbers
	n	constant size
	T	size type variable
	b	power of two size
	s/b	dyadic number size
	$s + s'$	addition of sizes
	$s - s'$	subtraction of sizes
	$s * s'$	multiplication of sizes
$range, R$	$::=$	range of numeric indices
	$s..s'$	Range with unit step
	$s, b..s'$	Range with step power of two
$tdecl, TV$	$::=$	Type variable declaration

		T	Unconstrained type variable
		$T : \mathbf{Field}$	Field type variable
		$T : \mathbf{Scalar} \ T'$	Scalar field of group
		$T : \mathbf{Group}$	Group type variable
		$T : \mathbf{Pairing} \ T_1 \ T_2$	Pairing-friendly groups
		$T : \mathbf{Size}$	Dyadic size type variable
		$T : \mathbf{Bin}$	Exponent size type variable
$type, \ t$	$::=$		Static types
		$\mathbf{Uni}\langle T, s \rangle$	Univariate polynomial
		$\mathbf{Mle}\langle T, s \rangle$	Multilinear polynomial
		$[t; s]$	Vector type
		T	Base type
$tdecls, \ TVs$	$::=$		Type variable declarations
		TV_0, \dots, TV_n	Type variables
$principal, \ P$	$::=$		Principal type
		T	Principal identifier
$principals, \ Ps$	$::=$		Principal types
		P_0, \dots, P_n	Principal identifiers
arg	$::=$		Argument
		$x : t$	Typed argument
		x	Untyped argument
		$\mathbf{public} \ x$	Untyped public argument
		$\mathbf{public} \ x : t$	Typed public argument
		$\mathbf{private}\langle Ps \rangle \ x$	Untyped private argument
		$\mathbf{private}\langle Ps \rangle \ x : t$	Typed private argument
$args$	$::=$		Arguments
		arg_0, \dots, arg_n	Argument list
$assert$	$::=$		Zippel assertion
		$\mathbf{assert}\langle P \rangle \ bexp$	Assertion
$exp, \ e, \ d$	$::=$		Zippel expressions
		f	Field literal
		x	Variable
		$e + d$	Addition
		$e - d$	Subtraction
		$e * d$	Product
		$e \cdot d$	Inner product
		e / d	Division
		$e \wedge d$	Exponentiation
		$\mathbf{gen}\langle T \rangle$	Group generator
		$\mathbf{coef}(e)$	Coefficients of univariate polynomial
		$\mathbf{mle}(e)$	Evaluations of MLE over boolean hypercube
		$\mathbf{interpolate}(e, d)$	Lagrange interpolation over evaluations
		$\mathbf{challenge}\langle t \rangle$	Generate a challenge of given type
		$[e_0, \dots, e_n]$	Vector of expressions

		$e ++ d$	Vector concatenation
		$[\mathbf{map} \ e \ \mathbf{for} \ x \ i_n \ \mathbf{r}]$	Map comprehension
		$x[s]$	index a vector
		$x[R]$	slice a vector
		$e(d_0, \dots, d_n)$	Function application
		$\mathbf{assert}; e$	Assertion-guarded expression
		$\mathbf{let} \ arg := e \ i_n \ d$	Let binding
		$arg \leftarrow e; d$	Transcript binding
$bexp$	$::=$		Boolean expressions
		$e = e'$	Equality of expressions
		$bexp \ \&\& \ bexp'$	Boolean and operation
		$bexp \ \ bexp'$	Boolean or operation
$decl, \ D$	$::=$		Top-level declarations
		$\mathbf{proto} \langle Ps \rangle \ \mathbf{ident} \ \langle TVs \rangle \ (args) \ \mathbf{where} \ bexp := e; \mathbf{assert}$	
		$\mathbf{fn} \ \mathbf{ident} \langle TVs \rangle (args) \rightarrow t := e$	

CHAPTER 6

Discussion

This dissertation has explored the development of expressive specification languages and their corresponding proof systems, demonstrating how formal verification can scale to handle increasingly complex program behaviors while maintaining modularity and machine-checkability. Our journey began with finite trace specifications and culminated in rich temporal properties of concurrent systems and zero-knowledge protocols, each stage building upon and extending previous foundations.

We started by addressing the challenge of specifying properties of finite program traces through SAFA, our novel extension to alternating finite automata. SAFA’s key innovation lies in its ability to efficiently handle extended regular expressions with wildcards, enabling specifications to focus on critical program behaviors while safely skipping irrelevant trace segments. The combination of SAFA’s multithreaded verification algorithm and its compact proof objects makes it particularly well-suited for practical applications, as demonstrated by its integration into the Reef compiler for zero-knowledge cryptographic certificates.

Moving beyond finite traces, we developed Ticl to address the verification challenges posed by infinite programs with complex behaviors. Ticl represents a significant advancement in bridging the gap between program logics and temporal logics, offering a structural approach to verifying both safety and liveness properties. By internalizing complex (co-)inductive proof techniques into high-level structural lemmas, Ticl enables remarkably concise and modular proofs of sophisticated program behaviors. Our case studies—ranging from job queue systems to concurrent shared memory and distributed consensus protocols—demonstrate Ticl’s expressiveness and practical utility in verifying real-world systems.

The culmination of our work is the proposal of Zippel, a language that brings together our insights on specification expressiveness and proof modularity to address the challenges of implementing zero-knowledge protocols. Zippel’s approach is unique in combining strong static guarantees

through its type system with automatic verification of zero-knowledge properties via information flow techniques. This bridges a crucial gap between theoretical security properties and practical implementations, while maintaining the theme of machine-checked verification that runs throughout this dissertation.

Throughout this progression, several key themes emerge:

First, the power of abstraction mechanisms in enabling scalable verification. From SAFA’s ability to abstract away irrelevant trace segments to Ticl’s high-level structural lemmas and Zippel’s type-based security guarantees, each contribution demonstrates how appropriate abstractions can make complex verification tasks tractable.

Second, the importance of machine-checked proofs in providing robust correctness guarantees. All three systems produce verifiable proof objects or certificates, enabling independent validation of their results. This emphasis on checkability enhances the trustworthiness and transparency of verified systems while maintaining practical efficiency.

Third, the value of bridging theoretical foundations with practical implementations. Each contribution connects formal methods theory—whether from automata theory, temporal logic, or cryptography—with concrete implementation concerns, resulting in tools and techniques that are both theoretically sound and practically applicable.

Looking forward, this work opens several promising directions for future research. The techniques developed for SAFA could be extended to handle even richer classes of specifications, potentially incorporating ideas from Separation Logic. Ticl’s approach to structural verification could be adapted without much effort to verify bigger concurrent and distributed, non-terminating systems. The Zippel language could be honed to fit the growing needs of protocol authors, with more cryptographic primitives, while maintaining its strong safety guarantees.

In conclusion, this dissertation demonstrates that expressive specifications and modular, machine-checked proofs are not mutually exclusive goals. Through careful design of specification languages

and proof systems, we can scale formal verification to handle increasingly sophisticated program behaviors while maintaining the clarity and reliability that make formal methods valuable. As software systems continue to grow in complexity and criticality, such verified foundations will become increasingly essential for ensuring their correctness and security.

APPENDIX A

APPENDIX A

A.1. Preliminary definitions

A.1.1. Monoids

A monoid is a triple (A, \cdot, ϵ) where A is the carrier set, \cdot is the *append* operation, and ϵ is the identity element of append, such that the monoid equations apply.

- Associativity $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- Left-identity $\epsilon \cdot a = a$.
- Right-identity $a \cdot \epsilon = a$.

A.1.2. Boolean Algebras

A boolean algebra (or boolean lattice) is a 6-tuple $(A, \top, \perp, \wedge, \vee, \neg)$ where A is the carrier set, $\top \in A, \perp \in A$ represent the *true* and *false* booleans.

The binary combinators \wedge, \vee correspond to conjunction and disjunction respectively, and the unary \neg corresponds to negation. A boolean algebra is closed in A under \wedge, \vee, \neg and has the following equations.

- Associativity of \vee, \wedge .
- Commutativity of \vee, \wedge .
- Distributivity of \wedge over \vee and \vee over \wedge .
- \perp the unit of \vee .
- \top the unit of \wedge .
- Annihilation for \vee, \top and \wedge, \perp respectively.
- Idempotence of \vee, \wedge .
- Complement rules for \vee, \wedge and \neg .

A.1.3. Kleene Algebras

Kleene algebras provide an elegant algebraic framework for reasoning about regular expression behavior. A *Kleene Algebra* is the 6-tuple $(A, \perp, \vee, \cdot, \epsilon, *)$, where A is a set with a monoid operator (\cdot, ϵ) and a choice operator (\vee, \perp) . The closure operator repeats an expression 0 or more times, defined as

$$r^* = \epsilon \vee r \cdot r^*$$

.

The following laws apply, like commutativity and associativity (\vee) , distributivity and closure laws apply, on top of the boolean and monoid laws which we omit.

$$(a \vee b) \vee c = a \vee (b \vee c)$$

$$(a \vee b) = (b \vee a)$$

$$(a \vee \perp) = a$$

$$(a \vee a) = a$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

$$\epsilon \cdot a = a \cdot \epsilon = a$$

$$\perp \cdot a = \perp \cdot a = \perp$$

$$\epsilon^* = \epsilon$$

$$\perp^* = \epsilon$$

$$(a \vee b) \cdot x = (a \cdot x) \vee (b \cdot x)$$

$$x \cdot (a \vee b) = (x \cdot a) \vee (x \cdot b)$$

A.1.4. Extended Kleene Algebras

Extended Kleene algebras add the product \wedge, \top and negation \neg operators to Kleene Algebras, defining a Monoid structure combined with a Boolean Algebra. *Extended Kleene Algebra* over carrier set A is the 9-tuple $(A, \top, \perp, \wedge, \vee, \neg, \cdot, *, \epsilon)$.

Extended Kleene Algebras have the same laws as Kleene Algebras, with the addition of the fol-

lowing laws for associativity, commutativity and unit of \wedge , De Morgan's laws for negation \neg and distributivity for \wedge .

$$(a \wedge \top) = a$$

$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$

$$(a \wedge b) = (b \wedge a)$$

$$(a \wedge b) \cdot c = (a \cdot c) \wedge (b \cdot c)$$

$$(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$$

$$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$$

$$\top^* = \top$$

$$x \cdot (a \wedge b) = (x \cdot a) \wedge (x \cdot b)$$

$$x \cdot (a \vee b) = (x \cdot a) \vee (x \cdot b)$$

$$\neg\neg r = r$$

$$\neg\top = \perp$$

$$\neg\perp = \top$$

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\neg(a \vee b) = \neg a \wedge \neg b$$

BIBLIOGRAPHY

- [ac22] arkworks contributors. **arkworks** zksnark ecosystem, 2022.
- [AHK02] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5), 2002.
- [AHM⁺17] Danel Ahman, Cătălin Hrițcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.
- [AIM⁺24] Sebastian Angel, Eleftherios Ioannidis, Elizabeth Margolin, Srinath Setty, and Jess Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3801–3818, 2024.
- [AKMQ23] JP Aumasson, Dmitry Khovratovich, Bart Mennink, and Porçu Quine. SAFE: Sponge API for field elements. Cryptology ePrint Archive, Paper 2023/522, 2023.
- [AM01] Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5), 2001.
- [AMO19] Alessandro Artale, Andrea Mazzullo, and Ana Ozaki. Do you need infinite time?. In *IJCAI*, 2019.
- [Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
- [BBB⁺18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the 39th IEEE Symposium on Security and Privacy, S&P '18*, pages 315–334, 2018.
- [BCG88] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical computer science*, 59(1-2), 1988.
- [BCI⁺16] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*, pages 387–393. Springer, 2016.

- [BCL12] Gergei Bana and Hubert Comon-Lundh. Towards unconditional soundness: Computationally complete symbolic attacker. In *International Conference on Principles of Security and Trust*, pages 189–208. Springer, 2012.
- [BDG⁺12] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. *International School on Foundations of Security Analysis and Design*, pages 146–166, 2012.
- [Bow17] Sean Bowe. Bls12-381: New zk-snark elliptic curve construction. *Zcash Company blog*, URL: <https://z.cash/blog/new-snark-curve>, 22, 2017.
- [BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 626–643. Springer, 2012.
- [Brz64] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.
- [CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Proceedings of the 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT ’23*, pages 499–530, 2023.
- [CCK⁺20] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R Tuttle. Code-level model checking in the software development workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 11–20, 2020.
- [CCM14] Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. A general framework for the derivation of regular expressions. *RAIRO-Theoretical Informatics and Applications*, 48(3):281–305, 2014.
- [CET⁺24] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. Sok: What don’t we know? understanding security vulnerabilities in snarks, 2024.
- [CHH⁺23] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 7(POPL), 2023.
- [CHM⁺20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS.

In *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '20, 2020.

- [CKS81] Ashok K Chandra, Dexter C Kozen, and Larry J Stockmeyer. Alternation. *Journal of the ACM (JACM)*, 28(1):114–133, 1981.
- [DGV⁺13] Giuseppe De Giacomo, Moshe Y Vardi, et al. Linear temporal logic and linear dynamic logic on finite traces. In *Ijcai*, volume 13, 2013.
- [DMWG23] Quang Dao, Jim Miller, Opal Wright, and Paul Grubbs. Weak fiat-shamir attacks on modern proof systems. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 199–216. IEEE, 2023.
- [DNV90] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
- [DS16] Christian Doczkal and Gert Smolka. Completeness and decidability results for ctl in constructive type theory. *Journal of Automated Reasoning*, 56, 2016.
- [DSFG21] Emanuele D’Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 2021.
- [EC82] E Allen Emerson and Edmund M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3), 1982.
- [EH86] E Allen Emerson and Joseph Y Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.
- [FJY90] Abdelaziz Fellah, Helmut Jürgensen, and Sheng Yu. Constructions for alternating finite automata. *International journal of computer mathematics*, 35(1-4):117–132, 1990.
- [FS86] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings of the 6th Annual International Cryptology Conference*, CRYPTO '86, pages 186–194, 1986.
- [Gro11] Jens Groth. Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '11, pages 431–448, 2011.
- [Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the 35th Annual International Conference on Theory and Applications of Cryptographic*

Techniques, EUROCRYPT '16, pages 305–326, 2016.

- [Gur21] Kobi Gurkan. Plonk custom gates design considerations, 2021. <https://kobi.one/2021/05/20/plonk-custom-gates.html>.
- [GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* 2019/953, 2019.
- [Hal97] Joseph Y Halpern. A theory of knowledge and ignorance for many agents. *Journal of Logic and Computation*, 7(1):79–108, 1997.
- [HHK⁺15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2015.
- [Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5), 1997.
- [Hop71] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.
- [JKJ⁺18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.
- [KP84] Dexter Kozen and Rohit Parikh. A decision procedure for the propositional μ -calculus. In *Logics of Programs: Workshop, Carnegie Mellon University Pittsburgh, PA, June 6–8, 1983*. Springer, 1984.
- [KST22a] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.
- [KST22b] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Proceedings of the 42nd Annual International Cryptology Conference, CRYPTO '22*, pages 359–388, 2022.
- [KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT '10*, pages 177–194, 2010.
- [Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming*

Languages and Systems (TOPLAS), 16(3), 1994.

- [Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), 2001.
- [LCK⁺23] Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. Fair operational semantics. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 2023.
- [LF16] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 385–399, 2016.
- [LT93] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.
- [MAA⁺19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hrițcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proceedings of the ACM on Programming Languages*, 3(ICFP), 2019.
- [Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.
- [Mes08] José Meseguer. 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*, pages 354–382. Springer, 2008.
- [ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. iee, 1977.
- [Pou16] Damien Pous. Coinduction all the way up. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, 2016.
- [Sch91a] Claus P. Schnorr. Efficient signature generation by smart cards. 4(3):161–174, 1991.
- [Sch91b] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4:161–174, 1991.
- [Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the 40th Annual International Cryptology Conference, CRYPTO ’20*, pages 704–737, 2020.

- [SGG⁺21] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.
- [Sha21] Ax Sharma. Major bgp leak disrupts thousands of networks globally. <https://www.bleepingcomputer.com/news/security/major-bgp-leak-disrupts-thousands-of-networks-globally/>, 2021.
- [SMG⁺24] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Yu. Anvil: Verifying liveness of cluster management controllers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.
- [ST81] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981.
- [Ste21] Steve Trimble. Software bugs rattle u.s. air force f-35 block 4 progress. <https://aviationweek.com/defense/aircraft-propulsion/software-bugs-rattle-us-air-force-f-35-block-4-progress>, 2021.
- [SVW87] A Prasad Sistla, Moshe Y Vardi, and Pierre Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2-3), 1987.
- [SZ21] Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.
- [TB17] Gadi Tellez and James Brotherston. Automatically verifying temporal properties of pointer programs with cyclic proof. In *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 491–508. Springer, 2017.
- [The24] The Coq Development Team. The Coq reference manual – release 8.19.0. <https://coq.inria.fr/doc/V8.19.0/refman>, 2024.
- [Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6), 1968.
- [Var07] Moshe Y Vardi. Automata-theoretic model checking revisited. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 137–150. Springer, 2007.

- [WAH⁺22] Théo Winterhalter, Cezar-Constantin Andrici, C Hrițcu, Kenji Maillard, G Martínez, and Exequiel Rivas. Partial dijkstra monads for all. *TYPES*, 2022.
- [WTS⁺18] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943. IEEE, 2018.
- [XZH⁺19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2019.
- [XZH⁺20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees. *Proceedings of the ACM on Programming Languages*, 4, 2020.
- [YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced research working conference on correct hardware design and verification methods*. Springer, 1999.
- [YTGN24] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. Mostly automated verification of liveness properties for distributed protocols with ranking functions. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024.
- [ZXL20] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. A framework and dataset for bugs in ethereum smart contracts. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*, pages 139–150. IEEE, 2020.