Verified Symbolic Execution with Kripke Specification Monads (and no Meta-Programming)

ANONYMOUS AUTHOR(S)

Verifying soundness of symbolic execution-based program verifiers is a significant challenge. This is especially true if the resulting tool needs to be usable outside of the proof assistant, in which case we cannot rely on shallowly embedded assertion logics and meta-programming. The tool needs to manipulate deeply embedded assertions, and it is crucial for efficiency to eagerly prune unreachable paths and simplify intermediate assertions in a way that can be justified towards the soundness proof. Only a few such tools exist in the literature, and their soundness proofs are intricate and hard to generalize or reuse. We contribute a novel, systematic approach for the construction and soundness proof of such a symbolic execution-based verifier. We first implement a shallow verification condition generator as an object language interpreter in a specification monad, using an abstract interface featuring angelic and demonic nondeterminism. Next, we build a symbolic executor by implementing a similar interpreter, in a symbolic specification monad. This symbolic monad lives in a universe that is Kripke-indexed by variables in scope and a path condition. Finally, we reduce the soundness of the symbolic execution to the soundness of the shallow execution by relating both executors using a Kripke logical relation. We report on the practical application of these techniques in Katamaran, a tool for verifying security guarantees offered by instruction set architectures (ISAs). The tool is fully verified by combining our symbolic execution machinery with a soundness proof of the shallow verification conditions against an axiomatized separation logic, and an Iris-based implementation of the axioms, proven sound against the operational semantics. Based on our experience with Katamaran, we can report good results on practicality and efficiency of the tool, demonstrating practical viability of our symbolic execution approach.

CCS Concepts: • **Theory of computation → Logic and verification**; **Programming logic**; **Hoare logic**; **Separation logic**; **Program verification**.

Additional Key Words and Phrases: program verification, symbolic execution, predicate transformers, separation logic, refinement, logical relations

ACM Reference Format:
Anonymous Author(s). 2018. Verified Symbolic Execution with Kripke Specification Monads (and no Meta-Programming). *J. ACM* 37, 4, Article 111 (August 2018), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Program logics based on Hoare logic and separation logic allow the modular verification of a very general class of correctness properties of software, including memory safety, absence of race conditions, functional correctness, termination [Gotsman et al. 2009] etc. However, derivations in such program logics take the form of large proof trees that are unrealistic to construct by hand. Instead, verification tools are used which guarantee the existence of a Hoare logic proof on successful verification. These tools use techniques like symbolic execution (SE) or weakest preconditions (WP) to decide largely automatically whether a program satisfies the program logic rules. For many tools like Dafny [Leino 2010], Frama-C [Kirchner et al. 2015], VeriFast [Jacobs et al. 2010] etc.[Ahrendt et al. 2014; Cohen et al. 2009; Filliâtre and Marché 2007; Leino et al. 2009] this

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

decision has to be taken on faith: verification can be trusted only if the verification tool is assumed to be bug-free.

Stronger assurance is provided by verified tools like VST [Cao et al. 2018] or Bedrock [Chlipala 2011]. These tools are implemented in a proof assistant like Coq or Isabelle and come with a mechanically verified soundness proof. Such a proof guarantees that whenever the tool succesfully verifies a program, there must exist a valid program logic derivation proving the program correct. However, implementing and proving soundness of a verification tool is a challenging task.

Part of the complexity stems from the need to avoid combinatorial explosion. Naive implementations of both WP and SE, lead to exponential blowup which can be observed in practice [Flanagan and Saxe 2001]. There are multi-stage approaches relying on program transformations to generate WPs quadratic in size in terms of the input program [Flanagan and Saxe 2001; Leino 2005]. However, there is only little work on verified implementations [Parthasarathy et al. 2021; Vogels et al. 2010], the program transformation impairs debugability of verification failures, and it is unclear how this approach scales to other assertion logics like separation logic. Symbolic executors try to counter the exponential explosion by explicitly manipulating assertions representing intermediate states (path constraints, symbolic heaps etc.), symbolically simplifying states [Cadar et al. 2008; Visser et al. 2012] and eagerly prune unreachable states by calling into automated theorem provers during execution to detect unsatisfiable constraints [Cadar et al. 2008; Jacobs et al. 2010].

Many verified tools address this challenge similarly to the SE approach, by shallowly embedding intermediate assertions as meta-logic properties and by making use of the meta-logic's meta-programming facilities to conveniently implement assertion simplification and pruning [Cao et al. 2018; Charguéraud 2010, 2011; Chlipala 2011; Chlipala et al. 2009]. However, this makes the tools depend on meta-programming languages like Coq's Ltac for their execution. Practically, this precludes the use of meta-languages' program extraction facilities, making it impossible to use the verification tool outside of the proof assistant interpreter. This makes it hard to offer an easy-to-use verification interface for users without experience with proof assistants, or interface with external tools like witness-producing SMT solvers. Additionally, executing a verifier in Coq's interactive interface can be significantly slower than executing an extracted version.

Implementing a sound verification tool *without* the use of meta-programming complicates an already considerable challenge further. It requires a deep embedding of program logic assertions and careful book-keeping of logic variables in scope, while preserving the guarantee that the intermediate assertions accurately represent all possible program paths. Only two existing SE-based tools have managed this: VeriSmall [Appel 2011] and Featherweight VeriFast [Jacobs et al. 2015]. However, the former features a restricted assertion language and the latter has a soundness proof that is intricate and hard to generalize to other tools (see Section 7 for a more detailed comparison).

In this paper, we contribute a new, systematic approach for constructing a sound program logic verifier, parametrized by a theory of user-implemented assertions, and lemmas. In more detail, we make the following contributions:

- We show how to implement symbolic verification condition (VC) generators by writing interpreters in symbolic predicate transformer monads.
- We define a Kripke frame of path constraints and logical variable contexts: a mathematical structure that we use to make contextual information (path constraints and logical variables) available to locally prune infeasible paths during symbolic execution.
- We demonstrate how eager solution of variable equalities and pruning of unreachable paths can be implemented modularly, and combined with a postprocessing of the VC tree resulting from symbolic execution to obtain simple VCs.

- We show how symbolic VC generation can be proved sound w.r.t. shallow VC generation by means of a novel logical relation.
- We compare efficiency of our implementation against related work and report some measurements that provide more insight in the performance characteristics of our approach.
- We demonstrate the reusability of our approach by implementing VC generators for μSail. The latter forms the basis for Katamaran, a verified tool for verifying security guarantees offered by instruction set architectures, whose semantics is defined in the Sail language.

We explain our approach step by step, starting in Section 2 with a shallow verification condition generator, that translates a statement to a Coq proposition. It is implemented as a monadic interpreter in a specification monad. Next, Section 3 presents a symbolic executor that produces a deeply embedded verification condition, implemented as a very similar monadic interpreter in a symbolic specification monad. The section also explains how the symbolic executor prunes unreachable paths and simplifies assertions during execution. Section 4 extends the two executors to verify separation logic. Section 5 establishes soundness of the symbolic verification conditions relative to the shallow ones, by constructing a Kripke-indexed logical relation between the two specification monads. Finally, Section 6 explains Katamaran, a verified separation logic verifier based on the techniques of this paper, to verify security properties of ISAs.

2 SHALLOW VC GENERATION

A verification condition is a formula whose validity is sufficient for the correctness of a program w.r.t. its specification. The traditional way to generate VCs is based on Dijkstra's weakest precondition calculus: we reduce validity of a Hoare triple to a first-order formula obtained by calculating the weakest (liberal) precondition of the given postcondition:

$$\{\mathrm{P}\}\ s\ \{\mathrm{Q}\} \leftrightarrow (\mathrm{P} \to \mathrm{wp}\ s\ \mathrm{Q})$$

The weakest precondition operator maps a statement s to a predicate transformer, which in turn is a mapping from predicates on the output state (postcondition) to a predicate on the input state. Or more generally, for a computation with input I and output O, the wp predicate transformer is of type $\mathrm{Pred}\ O \to \mathrm{Pred}\ I$. An important realization is that such predicate transformer types are monads [Ahman et al. 2017; Jacobs 2014; Swamy et al. 2013].

Indeed, taking $\mathrm{Pred}\ x := x \to \mathbb{P}$ and rearranging the input type I, the type above is nothing more than a mapping to the continuation monad with result type $\mathbb{P}$

$$I \to (O \to \mathbb{P}) \to \mathbb{P}$$

also called the *backwards predicate transformer monad* [Maillard et al. 2019].

This has been exploited in the F* language [Swamy et al. 2016] to index effectful monadic computations with their semantics as predicate transformers, allowing the user to co-design programs and specifications. Furthermore, [Maillard et al. 2019] generalizes this to other effects like state, exceptions and non-determinism simply by applying monad transformers to the base monad $W_{\mathrm{pure}}\ x := (x \to \mathbb{P}) \to \mathbb{P}$. Since these monads are used to define the specification rather than the implementation of functions, they are called *specification monads*.

Under this view we can see the wp as being a monadic interpreter [Liang et al. 1995] for our object language, with the result of the interpreter being the weakest precondition semantics for an object language expression.

In the remainder of this section, we develop such a monadic interpreter for the object language in Fig. 1, which is a simplified version of μSail, the language that Katamaran works with (see Section 6). Our intention is to implement such interpreters in the internal language (our host language) of a theorem prover, to yield a VC generator for the object language that produces VCs

$$
\begin{array}{lll}
v & ::= & n \mid \text{true} \mid \text{false} \mid \text{inl } v \mid \text{inr } v \mid (v, v) \mid v :: v \mid [] \mid () \\
e & ::= & x \mid n \mid \text{true} \mid \text{false} \mid \text{inl } e \mid \text{inr } e \mid (e, e) \mid e :: e \mid [] \mid () \mid e; e \mid e\ op\ e \\
 & & \mid \textbf{let } x := e \textbf{ in } e \mid x := e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \textbf{call } f\ \overline{e} \mid \textbf{case } e \textbf{ of } (x, x) \Rightarrow e \mid \ldots \\
op & ::= & + \mid = \mid \leq \mid \ldots \\
prg & ::= & \overline{f\ \overline{x} := e} \\
\delta & ::= & \overline{x \mapsto v}
\end{array}
$$

Fig. 1. Object language syntax

represented directly as propositions in the host language, i.e. a shallow embedding. The next section focuses on deep embeddings, i.e. symbolic representations.

Fig. 1 defines the grammar of values v, expressions e, arithmetic, relational and boolean operators op and programs prg. We use an overline to denote a repetition, i.e. a program is a list of definitions of functions f which in turn each have a list of formal parameters $\overline{x}$. Otherwise, the grammar follows a standard presentation for boolean, integer, sum, product, list and unit types. As a notational convention we use **case**...**of**... to denote pattern matching on structural types in the object language and **match**...**with**... for pattern matching in the host language.

Our running example is the following function that computes the sum, max and length of a list, adapted from the first Verified Software Competition [Klebanov et al. 2011]: For readability, we include a type signature:

```
summaxlen (xs : list int) : (int * int) * int :=
  case xs of
  | [] ⇒ ((0, 0), 0)
  | y :: ys ⇒ let sml := call summaxlen ys in
              case sml of | (sm, l) ⇒ case sm of | (s, m) ⇒
                  ((s + y, if m < y then y else m), l + 1)
```

The particular monad that we use for the interpreter is

$$W_\delta\ x := (x \to \delta \to \mathbb{P}) \to \delta \to \mathbb{P}$$

which is obtained by transforming W_{pure} with the state transformer with state type δ, and uncurrying the result. δ is a local variable store, i.e. a mapping of program variables to values. The signature of the interpreter on expressions is $\text{exec} : e \to W_\delta\ v$, and we reserve wp for informal discussions about predicate transformers.

2.1 Control-flow branching

The interpreter exec can be written in different ways and it is important to consider the consequences of different styles. For example, we could implement the interpretation of an if expression as follows:

$$\text{exec } (\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2) = v \leftarrow \text{exec } e; \textbf{if } v \textbf{ then exec } e_1 \textbf{ else exec } e_2$$

Such a definition yields a semantically adequate predicate transformer, but it results in weakest preconditions that use host language pattern matching (hidden in the host language **if**-statement). Such a wp calculation works for producing shallowly embedded VCs, but it can block when pattern matching (using **if**, in this case) on universally or existentially quantified values.

Instead, our interpreter expresses control-flow branching using propositional features. As we will see in Section 3, this will make it easier to manipulate the resulting postconditions when we extend our approach to a symbolic interpreter. As an example consider the traditional weakest

$$
\begin{aligned}
\text{angelic} : W_\delta\ v &:= \lambda\ post\ \delta.\ \exists v.\ post\ a\ v \\
\text{demonic} : W_\delta\ v &:= \lambda\ post\ \delta.\ \forall v.\ post\ a\ v \\
m_1 \oplus m_2 : W_\delta\ a &:= \lambda\ post\ \delta.\ m_1\ post\ \delta \vee m_2\ post\ \delta \\
m_1 \otimes m_2 : W_\delta\ a &:= \lambda\ post\ \delta.\ m_1\ post\ \delta \wedge m_2\ post\ \delta \\
\text{assert}(p : \mathbb{P}) : W_\delta\ () &:= \lambda\ post\ \delta.\ p \wedge post\ ()\ \delta \\
\text{assume}(p : \mathbb{P}) : W_\delta\ () &:= \lambda\ post\ \delta.\ p \longrightarrow post\ ()\ \delta
\end{aligned}
$$

Fig. 2. Primitives for angelic and demonic non-determinism, assumptions and assertions in the specification monad with State.

precondition rule for if-conditionals of which there are two variants

$$
\begin{aligned}
& wp(\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ P \\
& \leftrightarrow wp\ e\ \ (\lambda v.\ (v = \textbf{true} \longrightarrow wp\ e_1\ P) \wedge (v = \textbf{false} \longrightarrow wp\ e_2\ P)) \\
& \leftrightarrow wp\ e\ \ (\lambda v.\ (v = \textbf{true} \wedge wp\ e_1\ P) \vee (v = \textbf{false} \wedge wp\ e_2\ P))
\end{aligned}
$$

which use conjunction and implication resp. disjunction and conjunction. Adopting this approach in our interpreter means that we effectively stop relying on the shallow embedding and can always normalize wp computations to a sub-language of the propositions. We will make this more precise in Section 3 where we define a language of deeply embedded propositions.

We can express the logical connectives at a higher-level of abstraction to hide the plumbing: they correspond to angelic and demonic non-determinism features of the monad and guards for local assumptions and assertions [Dijkstra 1975; Jacobs et al. 2015; Nelson 1989]. Fig. 2 presents the definitions.

The guard assume indicates that subsequent computations may assume the validity of a given proposition. Conversely, assert indicates that a given proposition is required to hold.

The binary angelic choice $m_1 \oplus m_2$ non-deterministically chooses between two subcomputations, but it is sufficient for one of them to succeed in order for the combined computation to succeed. Binary demonic choice $m_1 \otimes m_2$ works similarly, but both subcomputations must succeed in order for the combined computation to succeed. The terminology is based on the intuition that the non-deterministic choice is made by a sympathetic resp. adversarial party. The same distinction exists between operators angelic and demonic which non-deterministically produce an object language value. This can be generalized to arbitrary host language values, but our interpreter only needs it for v.

The primitives in Fig. 2 allow us to elegantly define our interpreter but at the same time avoid host language pattern matching in the resulting weakest preconditions. Consider, for example, Fig. 3, where we show the interpretation of if expressions and pattern matching for sum types. The definition uses a function $\text{matchbool}_\otimes$ which uses $\otimes$ and assume to implement pattern matching on booleans. Pattern matching on sum types is implemented similarly, except that we additionally use demonic to choose values for x and y in the branches. We leave it as an exercise to the reader to verify that this definition of weakest preconditions for if expressions unfolds to one of the more traditional definitions mentioned above.

Note that traditionally, computing the wp for a statement s is regarded as executing s backwards starting from the postcondition. This view is understandable when we choose a first-order representation of predicates and choose a call by value evaluation strategy, i.e. when calculating $wp\ s\ Q$, the postcondition Q is normalized first before continuing the recursion over the statement s.

$$
\begin{aligned}
&\mathit{matchbool}\otimes\ v\ (m_1 m_2 : W_\delta\ a) : W_\delta\ a := (\mathit{assume}\ (v = \mathbf{true}); m_1) \otimes (\mathit{assume}\ (v = \mathbf{false}); m_2)\\
&\mathit{matchbool}\oplus\ v\ (m_1 m_2 : W_\delta\ a) : W_\delta\ a := (\mathit{assert}\ (v = \mathbf{true}); m_1) \oplus (\mathit{assert}\ (v = \mathbf{false}); m_2)\\
&\mathit{matchsum}\otimes\ v_0\ (f g : v \to W_\delta\ a) : W_\delta\ a := \quad (v_1 \leftarrow \mathit{demonic}; \mathit{assume}\ (v_0 = \mathbf{inl}\ v_1); f\ v_1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \otimes (v_1 \leftarrow \mathit{demonic}; \mathit{assume}\ (v_0 = \mathbf{inr}\ v_1); g\ v_1)\\
&\mathit{exec}\ (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) := b \leftarrow \mathit{exec}\ e_1;\ \mathit{matchbool}\otimes\ b\ (\mathit{exec}\ e_2)\ (\mathit{exec}\ e_3)\\
&\mathit{exec}\ (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}\ x \to el \mid \mathbf{inr}\ y \to er) :=\\
&\quad v \leftarrow \mathit{exec}\ e;\ \mathit{matchsum}\otimes\ v\ (\lambda vl.\ \mathit{push}\ (x, vl);\ v \leftarrow \mathit{exec}\ el;\ \mathit{pop};\ \mathit{return}\ v)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda vr.\ \mathit{push}\ (x, vr);\ v \leftarrow \mathit{exec}\ er;\ \mathit{pop};\ \mathit{return}\ v)
\end{aligned}
$$

Fig. 3. Weakest precondition for control-flow branches

$$
\begin{aligned}
\mathit{exec}\ (x := e) : W_\delta\ () &:= \mathit{eval}\ e \gg\!= \mathit{assign}\ x\\
\mathit{eval}\ e : W_\delta\ v &:= \lambda\delta\ \mathit{post}.\ \mathit{post}\ (\mathit{eval}_{exp}\ e\ \delta)\\
\mathit{assign}\ x\ v : W_\delta\ () &:= \lambda\delta\ \mathit{post}.\ \mathit{post}\ (\mathit{update}\ \delta\ x\ v)\ ()
\end{aligned}
$$

Fig. 4. Weakest precondition for mutable variable assignments

However, due to the higher-order representation of predicates in W_δ, recursion over e is happening first and the postcondition is only used at the *leaves* of exec e. This also entails that the resulting precondition is constructed from the root and effectively the execution proceeds in a forward fashion. Using continuations to construct trees from the root is an old trick [Hinze 2012; Hughes 1986; Hutton et al. 2010; Voigtländer 2008].

2.2 Assignment

For verifying assignments to mutable variables, we work in a specification monad with a mutable environment. Mutable variable assignments can be implemented as defined in Fig. 4. This replaces the traditional substitution $P[x \mapsto e]$ in the wp rule for assignments. In essence, we build up an explicit substitution [Abadi et al. 1991] that is lazily accumulated and forced once over P, i.e. when applying P to the store. This happens here implicitly, but can also be modelled explicitly in an implementation strategy, as for example in the KeY project [Ahrendt et al. 2014]. Importantly, execution of assignments also proceeds in a *forward fashion*, instead of being dependent on the result of a backwards running wp sub-calculation.

The definition in Fig. 4 can result in a size explosion. For example, an assignment like $x := x + x; x := x + x; x :=$ will produce 8 copies of the initial contents of the program variable x. For this paper, we ignore this issue but a usual trick is to introduce abbreviations using quantification:

$$\mathit{assign}'\ x\ v := v' \leftarrow \mathit{angelic}; \mathit{assert}\ (v = v'); \mathit{assign}\ x\ v'$$

or to canonicalize terms during symbolic execution.

2.3 Functions

To verify functions, we need a way to declare their specifications. For this, we define a form of contracts that specify pre- and postconditions. In the anticipation of moving to deep embeddings

$$V ::= \ell \mid n \mid \text{true} \mid \text{false} \mid \text{inl}\ V \mid \text{inr}\ V \mid (V, V) \mid V\ op\ V \mid V :: V \mid [] \mid ()$$
$$\Sigma ::= \overline{\ell} \qquad \zeta ::= \overline{\ell \mapsto V} \qquad \iota ::= \overline{\ell \mapsto v}$$

Fig. 5. Deeply-embedded values

in the next section, we already define parts of contracts in a symbolic way, and refine them in the next section.

Fig. 5 contains the basic definitions of our symbolic representation. To distinguish the definitions from their shallow counterparts, we use capital letters. Values V now contain a production for a logic variable ℓ that represents an abstract concrete value, that is for instance introduced by quantification. We treat logic variables as being in a namespace entirely separate from program variables x. Moreover, V contains all productions of e that we do not translate into predicates such as the operator production. We will interchangably refer to V as *symbolic values*, or *symbolic terms*. Furthermore, Fig. 5 defines logic contexts Σ, valuations ι (mappings from logic variables to concrete values), and substitutions ζ (mappings from logic variables to symbolic terms). To refer to the valuations of a particular logic context Σ, we use the notation $\iota_\Sigma := \{\overline{\ell \mapsto v} \mid \Sigma = \overline{\ell}\}$. We denote instantiation of a symbolic value to a concrete one by $V[\iota]$ and the application of a substitution by $V[\zeta]$.

The contract for a function $f\ \overline{x}$ is given by the production

contract ::= VARS Σ PAT $\overline{x \mapsto V}$ REQ $(\iota_\Sigma \to \mathbb{P})$ RES ℓ ENS $(\iota_{\Sigma,\ell} \to \mathbb{P})$

We view all contracts to be universally quantified over a given logic context Σ. The remainder specifies a Hoare triple for calling f. The arguments are specified by patterns PAT, one for each formal parameter $\overline{x}$. Patterns are represented as symbolic terms with free variables in Σ. This scheme avoids conflating logic and program variables, which would be necessary to specify contracts with program variables + ghost variables. It also gives us more flexibility in the form of non-linear patterns, i.e. logic variables can occur more than once in patterns. Usually, the pattern for a program variable is simply a logic variable. RES specifies a logic variable ℓ for the result value. The precondition REQ is a predicate on Σ-valuations and the postcondition a predicate on (Σ, ℓ)-valuations. In the following, we will skip the repetition of the formal parameters $\overline{x}$ and also refer to a contract more succinctly as being a tuple $(\Sigma, \overline{V}, req, \ell, ens)$. Overall, the contract encodes the following Hoare triple:

$$\forall \iota_\Sigma.\ \{req\ \iota_\Sigma\}\ \textbf{call}\ f\ \overline{V[\iota_\Sigma]}\ \{v.\ ens\ (\iota_\Sigma, \ell \mapsto v)\}$$

Coming back to our summaxlen example, we formulate the following contract: the computed sum should be less than or equal to the product of the computed maximum and length.

$$\forall xs : \text{list int}.\ \{\top\}\ summaxlen(xs)\ \left\{ res.\ \begin{array}{l} \textbf{match}\ res\ \textbf{with}\ (sm, l) \to \\ \quad \textbf{match}\ sm\ \textbf{with}\ (s, m) \to s \le m * l \wedge 0 \le l \end{array} \right\}$$

Function calls are verified by interpreting the contract as a specification statement [Morgan 1988], as defined in Fig. 6.

A function $f\ \overline{x} := e$ satisfies $(\Sigma, \overline{x \mapsto V}, req, res, ens)$ when the specified triple is valid for the body:

$$\forall \iota_\Sigma.\ req\ \iota_\Sigma \to \text{exec}\ e\ (\lambda\_\ v_{res}.\ ens(\iota_\Sigma, res \mapsto v_{res}))\ (\overline{x \mapsto V[\iota_\Sigma]})$$

This VC quantifies universally over a valuation for the logic variables of the contract. Next we assert that the instantiated precondition $req\ \iota_\Sigma$ implies the weakest precondition of the body as

$$
\begin{array}{l}
\mathit{exec}\ (\mathbf{call}\ f\ \overline{v}) := \\
\quad \text{let}\ (\Sigma, \overline{V}, \mathit{req}, \mathit{res}, \mathit{ens}) := \text{contract}\ f\ \text{in} \\
\quad \iota_\Sigma \leftarrow \mathit{angelic}{*}\ \Sigma;\ \overline{\mathit{assert}\ (v = V[\iota_\Sigma])};\ \mathit{assert}\ (\mathit{req}\ \iota_\Sigma); \\
\quad v_{res} \leftarrow \mathit{demonic};\ \mathit{assume}\ (\mathit{ens}\ (\iota_\Sigma, \mathit{res} \mapsto v_{res}));\ \mathit{pure}\ v_{res}
\end{array}
$$

Fig. 6. Weakest precondition for function calls

$$
\frac{\begin{array}{c}\{P\}\ e; \delta\ \{R\} \\ \forall \delta'.\ \{R\ \text{true}\ \delta'\}\ e_1; \delta'\ \{Q\} \\ \forall \delta'.\ \{R\ \text{false}\ \delta'\}\ e_2; \delta'\ \{Q\}\end{array}}{\{P\}\ \text{if}\ e\ \text{then}\ e_1\ \text{else}\ e_2; \delta\ \{Q\}}
\qquad
\frac{P \vdash P' \qquad \forall v, \delta'.\ Q'\ v\ \delta' \vdash Q\ v\ \delta' \qquad \{P'\}\ s; \delta\ \{Q'\}}{\{P\}\ s; \delta\ \{Q\}}
$$

$$
\frac{\forall a.\ \{P\ a\}\ e; \delta\ \{Q\}}{\{\exists a.\ P\ a\}\ e; \delta\ \{Q\}}
\qquad
\frac{\{P\}\ e; \delta\ \{\lambda\ v\ \delta'.\ Q\ v\ (\delta'[x \mapsto v])\}}{\{P\}\ x := e; \delta\ \{Q\}}
$$

$$
\frac{\begin{array}{c}\text{contract}\ f = (\Sigma, \overline{V}, \mathit{req}, \mathit{res}, \mathit{ens}) \qquad P \vdash \mathit{req}\ \iota_\Sigma \\ v = V[\iota_\Sigma] \qquad \forall v_{res}.\ \mathit{ens}\ (\iota_\Sigma, \mathit{res} \mapsto v_{res}) \vdash Q\ v_{res}\end{array}}{\{P\}\ \mathbf{call}\ f\ \overline{v}; \delta\ \{\lambda\ v_{res}\ \delta'.\ Q\ v_{res} \wedge (\delta = \delta')\}}
$$

Fig. 7. An excerpt of the program logic axioms that we prove our concrete executor sound against.

calculated by exec. The input store contains the formal parameters $\overline{x}$ mapping to the instantiated pattern $\overline{V[\iota_\Sigma]}$. We ignore the output store and pass the extended valuation to the postcondition *ens*.

However, for the purpose of automation, we use the following equivalent formulation in terms of assume and assert.

$$
\begin{array}{l}
\mathrm{vc}\ (f\ \overline{x} := e) : \mathbb{P} := \mathbf{let}\ (\Sigma, \overline{x \mapsto V}, \mathit{req}, \mathit{res}, \mathit{ens}) := \text{contract}\ f\ \mathbf{in} \\
\qquad \forall \iota_\Sigma.\ \mathbf{let}\ m : W_\delta\ () := \begin{cases}\text{assume}(\mathit{req}\ \iota_\Sigma); \\ v \leftarrow \text{exec}\ e; \\ \text{assert}(\mathit{ens}\ (\iota_\Sigma, \ell \mapsto v))\end{cases} \\
\qquad\qquad \mathbf{in}\ m\ (\lambda\_\ \delta'.\top)\ (\overline{x \mapsto V[\iota_\Sigma]})
\end{array}
$$

When we move to a symbolic executor in Section 3, this definition will allow us to add constraints from the precondition to the path constraints before executing the body and thereby prune more paths. This technique is also known as *preconditioned symbolic execution* [Baldoni et al. 2018]. Similarly, this lets the executor try to solve obligations resulting from the postcondition automatically.

2.4 Soundness

The verification conditions that we describe in this section can be proven sound with respect to an axiomatised program logic with judgements on configurations of the form $\{P\}\ e; \delta\ \{Q\}$, where δ is a store that assigns values to program variables. We make the store explicit and use regular propositions instead of using store predicates, since that makes the development of the program

logic much shorter. This is not a problem since its primarily used as part of the proof and not intended to be used directly. P and Q are the pre- resp. postcondition, the latter taking the result of s and the updated local store as an argument. The program logic is assumed to satisfy a number of axioms based on the ones used by [Cao et al. 2018] and [Malecha 2015]. An excerpt of the axioms is depicted in Fig. 7, including a standard triple for if statements, a standard consequence rule, a structural rule for eliminating an existentially quantified precondition and standard triples for assignments. We present a simplified version for function invocations, because our actual implementation requires A-normal form for function calls. We do not go into much detail on these axioms because they are standard and uncontroversial and in fact, we have proven them sound using an Iris [Jung et al. 2018] model against the operational semantics of μSail.

In terms of this program logic, we prove soundness of shallowly executing program expressions in the following lemma and derive a corollary for the soundness of shallow verification conditions.

Lemma 2.1 (Soundness of shallow execution). *If $\mathit{exec}\ e\ \mathit{post}\ \delta$ holds for an expression e, postcondition post and local store δ, then the following program logic triple holds:*

$$\{\top\}\ e; \delta\ \{\mathit{post}\}.$$

Alternatively phrased, the following triple always holds:

$$\{\mathit{exec}\ e\ \mathit{post}\ \delta\}\ e; \delta\ \{\mathit{post}\}.$$

Corollary 2.2 (Soundness of shallow verification condition generation). *If the verification conditions $\mathit{vc}\ f$ holds for for a function $f\ \overline{x} := e$ with contract $(\Sigma, \overline{V}, \mathit{req}, \ell, \mathit{ens})$, then the contract encodes a valid triple for the body of f, i.e. the following holds:*

$$\mathit{vc}\ f \longrightarrow \forall \iota_\Sigma.\ \{\mathit{req}\ \iota_\Sigma\}\ e; \overline{V[\iota_\Sigma]}\ \{\lambda\ v\ \delta'.\ \mathit{ens}\ (\iota_\Sigma, \ell \mapsto v)\}.$$

An important lemma that needs to be proved first is that all predicate transformers generated by the interpreter are monotonic. The soundness result is important, but despite the higher-order encoding of exec it is proven similarly to existing textbook proofs in the literature for the soundness of weakest preconditions [see, e.g., Nielson and Nielson 2007]. Specifically, FVF [Jacobs et al. 2015] shows the soundness of a shallow executor written in a specification monad against a concrete interpreter. So, we do not go into it in much detail. Instead, in the next sections, we explain our novel approach to proving symbolic execution soundness.

3 SYMBOLIC SPECIFICATION MONADS

The VC generated by the interpreter of the last section reflects the recursive structure of the execution and can be seen as a (symbolic) execution tree in which we only kept control-flow constraints, and assumptions and assertions coming from specifications, but removed transient execution state like the local variable store etc. Fundamentally, we cannot inspect shallow propositions in the interpreter itself. As a consequence, the shallow VCG will explore all execution paths through a function, without regard if this execution path is feasible given the precondition and the constraints imposed by control-flow branches. This is exacerbated in Section 4 where we implement a Smallfoot-style symbolic execution for separation logic, which makes heavy use of additional angelic non-determinism, but also adds new constraints during execution.

Ideally, we want to detect during calculation of the weakest-precondition when the execution paths become infeasible and discard this path entirely. For this, symbolic executors keep track of a path condition, the set of constraints leading to the current execution path, and use an automatic solver to detect when this path condition becomes inconsistent. In this section we develop an alternative implementation of our interpreter based on symbolic representations (deep embeddings) that achieves this. Note that the result of running this interpreter is still a proposition, albeit with

$$\mathbb{F} ::= \mathcal{P}\ \overline{V} \mid V = V \qquad \mathbb{C} ::= \overline{\mathbb{F}}$$
$$\mathbb{S} ::= \top \mid \bot \mid \mathbb{F} \to \mathbb{S} \mid \mathbb{F} \wedge \mathbb{S} \mid \mathbb{S} \wedge \mathbb{S} \mid \mathbb{S} \vee \mathbb{S} \mid \exists \ell.\mathbb{S} \mid \forall \ell.\mathbb{S} \mid \mathrm{debug}_D\ \mathbb{S} \mid (\ell \mapsto t) \to \mathbb{S} \mid (\ell \mapsto t) \wedge \mathbb{S}$$

Fig. 8. Deeply-embedded formulas and propositions

some sub-trees removed. More specifically, in the interpreter the path condition changes during *assume* and *assert* statements. Is an assumed formula inconsistent with the current path condition we can prune the path by emitting a true proposition $\top$. Is an asserted formula inconsistent with the path condition, we emit a false proposition $\bot$.

We will use the same machinery to automatically discharge proof obligations. These come from the postcondition of the function for which we calculate the VC and from the preconditions of called functions. This is achieved by developing a deeply-embedded assertion language for pre- and postconditions which is then also interpreted in our monad.

Technically, our symbolic interpreter is similar to the concrete interpreter from Section 2, except for three main changes. First, we use a specification monad that generates propositions in a symbolic universe of propositions $\mathbb{S}$ rather than meta-language propositions in $\mathbb{P}$. Additionally, the use of symbolic propositions requires careful bookkeeping of contextual information: the logic variables in scope and the path constraints. As we will see in this section, this contextual information defines the worlds of a Kripke frame. This means that we index values and computations with the world in which they are meaningful, allowing us to deal with this bookkeeping in a principled manner. This also enforces monotonicity of path constraints during execution by construction. Finally, thanks to the indexing, the current path constraints are always available during symbolic execution, and we interact with a constraint in order to eagerly simplify path conditions and prune unreachable paths.

In this section, we begin by defining a deep embedding of terms and formulas for path constraints in Section 3.1, an interface for a constraint solver in Section 3.2 and the mentioned Kripke frame in Section 3.3. Next, we combine these components in Section 3.4 in a symbolic specification monad to obtain a symbolic version of the VC generator from Section 2. We discuss the production of debug information in Section 3.5 and an additional simplification phase that in particular tries to instantiate quantifiers in Section 3.6. Finally, we finish with an example in Section 3.7.

3.1 Symbolic terms and constraints

In Section 2 we implemented angelic and demonic choice shallowly using existential and universal quantification from the meta-language. As a result, we could not programmatically inspect values or propositions during execution, for instance for semi-automatic simplification or solving. In this section, we develop a deep embedding instead. Fig. 8 contains definitions.

Basic symbolic formulas $\mathbb{F}$ represent exactly the propositions that are assumed or asserted during execution. There are two cases: first, all constraints introduced by control-flow branches are equalities between values, and second, pre- and postconditions that we consider belonging to an application-specific theory. We model these by parameterizing over a set $\mathcal{P}$ of application-specific predicates.

The result of our symbolic executor is a symbolic proposition $\mathbb{S}$ that contains all the necessary logical connectives: $\mathbb{F} \to \mathbb{S}$ resp. $\mathbb{F} \wedge \mathbb{S}$ are used for assume resp. assert, $\mathbb{S} \wedge \mathbb{S}$ and $\mathbb{S} \vee \mathbb{S}$ for binary choice, and $\exists \ell.\mathbb{S}$ and $\forall \ell.\mathbb{S}$ for arbitrary choice. We discuss the debug production in Section 3.5 and the last two productions at the end of the next section.

$$\text{solver} : \mathbb{C} \to \overline{\mathbb{F}} \to \text{option}(\zeta, \overline{\mathbb{F}})$$

$$\text{solver}\ \mathbb{C}\ \overline{\mathbb{F}_0} = \text{Some}(\overline{l \mapsto t}, \overline{\mathbb{F}_1}) \quad \Leftrightarrow (\mathbb{C} \vdash \overline{\mathbb{F}_0} \Leftrightarrow \mathbb{C} \vdash \overline{l = t} \wedge \overline{\mathbb{F}_1})$$

$$\text{solver}\ \mathbb{C}\ \overline{\mathbb{F}_0} = \text{None} \quad \Leftrightarrow \mathbb{C} \vdash \neg\overline{\mathbb{F}_0}$$

Fig. 9. Solver interface

3.2 Path constraints and entailment

To prune infeasible execution paths, we keep track of all basic formulas, that are assumed or asserted, in a path constraint $\mathbb{C}$. Each time a new formula $\mathbb{F}_0$ is added, we check for consistency and otherwise prune the path. More specifically, we call a solver with entailment queries of the form $\mathbb{C} \vdash \mathbb{F}_0$. Instead of a coarse satisfiability answer distinguishing three cases – yes, no, or undecided – we allow the solver to give back more information. Even in the undecided case, we can expect the solver to have made some progress, but eventually halted at a sub-problem $\overline{\mathbb{F}'}$, s.t. $\mathbb{C} \vdash \mathbb{F}_0 \leftrightarrow \mathbb{C} \vdash \overline{F'}$. We could therefore replace the original problem $\mathbb{F}_0$ with the sub-problem $\overline{\mathbb{F}'}$ when recording it in the final VC. This is useful since unsolved asserts are eventually presented to the user as proof obligations as part of proving the VC. Any simplification that we can already make fully automatically should be applied, to aid the user in proving the VC or to debug verification failures.

The execution of pattern matches introduces a lot of new logic variables and equality constraints. Ideally, we would like to simplify those automatically as well. To this end, we allow the solver also to report on possible unifications. We will use these later to instantiate existential quantifiers in Section 3.6. The resulting interface is in Fig. 9. The solver maps a list of formulas $\overline{\mathbb{F}_0}$ to a substitution (in triangular form [Baader et al. 2001]) $\zeta = \overline{\ell \mapsto V}$ and a list of residual formulas $\overline{\mathbb{F}_1}$ such that the entailment of the substitution ζ, seen as a system of equations, and the residual formulas is equivalent to the entailment of the original formula. We record such unifications in symbolic propositions using the special forms of assume $(\ell \mapsto t) \to \mathbb{S}$ and assert $(\ell \mapsto t) \wedge \mathbb{S}$. In both cases, the executor can eliminate the variable from further consideration by applying the substitution to all other data, e.g. the program variable store. A unification marks the end-of-scope [Hendriks and van Oostrom 2003] of the logic variable ℓ, i.e. we require $\mathbb{S}$ to be well-formed in $\Sigma - \ell$ in both productions.

3.3 Kripke frame and modal types

Implementing a symbolic executor poses some consistency challenges: an intermediate result, such as a symbolic term, formula or proposition, cannot be readily used under any path constraints other than the one they have been computed in. Using a value in a subset or a different set of constraints (ancestor or sibling in the execution tree) is unsound, but even using it under a larger set of constraints is not possible without further ado, since logic variables may have gone out of scope due to unifications. In this case, we have to apply the recorded substitutions first. To enforce consistent handling, we classify all values and computations with a set of logic variables and path constraints under which they are meaningful. This also helps us to structure the soundness proof in the next section.

Formally, Fig. 10 defines a set w, which we call worlds in the remainder, consisting of pairs of logic variable contexts Σ and path constraints that are well-formed under the given Σ. Worlds can be loosely identified with positions in the execution tree. The contained variable context Σ accumulates all existential and universal quantifiers on the path from that position to the root minus the variables that have been unified, and the path constraint collects all the assumed and

$$
\begin{array}{lll}
w & := \{(\Sigma, \mathbb{C}) \mid \mathrm{fv}(\mathbb{C}) \subseteq \Sigma\} & \\
(\Sigma_1, \mathbb{C}_1) \sqsubseteq (\Sigma_2, \mathbb{C}_2) & := \{\zeta \in \mathrm{Sub}[\Sigma_1, \Sigma_2] \mid \mathbb{C}_2 \vdash \mathbb{C}_1[\zeta]\} & \\
\hline
\vdash A & := \forall w, A\ w & \\
A \longrightarrow B & := \lambda w.A\ w \longrightarrow B\ w & \\
\Box A & := \lambda w. \forall w', w \sqsubseteq w' \longrightarrow A\ w' & \\
\hline
\mathbf{T} & : \vdash \Box A \longrightarrow A & := \lambda w\ a.a\ w\ \mathrm{id}_{\sqsubseteq} \\
\mathbf{4} & : \vdash \Box A \longrightarrow \Box\Box A & := \lambda w_0\ a\ w_1\ (\omega_1 : w_0 \sqsubseteq w_1)\ w_2\ (\omega_2 : w_1 \sqsubseteq w_2).a\ w_2\ (\omega_2 \circ_{\sqsubseteq} \omega_1) \\
\mathbf{K} & : \vdash \Box(A \longrightarrow B) \longrightarrow \Box A \longrightarrow \Box B & := \lambda w\ f\ a\ w'\ (\omega : w \sqsubseteq w').f\ w'\ \omega\ (a\ w'\ \omega) \\
\_[\_] & : \vdash A \longrightarrow \Box A & A \text{ persistent}
\end{array}
$$

Fig. 10. Kripke frame, modal types, and S4 axioms

asserted formulas with the unifications applied. To achieve the aforementioned classification, we work in the category of world indexed families $w \longrightarrow \mathrm{Type}$ rather than Type.

We will freely regard symbolic terms, stores, formulas, propositions, etc. as belonging to that universe by restricting them to the subset that is well-formed in the given world, e.g.

$$V := \lambda(\Sigma, \mathbb{C}).\{V \mid \mathrm{fv}(V) \subseteq \Sigma\}.$$

Similarly, the specification monad we define in this section will be a monad on $w \longrightarrow \mathrm{Type}$.

Fig. 10 also defines a preorder $\sqsubseteq$ between worlds, called the *accessibility relation*. Visually this (over)approximates the ancestor-descendant relationship in the execution tree, i.e. a world w_2 is accessible from a *base world* w_1 iff w_2 is a descendant (appears below) w_1 in the execution tree . Formally, we define that $(\Sigma_2, \mathbb{C}_2)$ is accessible from $(\Sigma_1, \mathbb{C}_1)$ iff there is a simultaneous substitution ζ for all logic variables in Σ_1 with symbolic terms in Σ_2, and under this substitution $\mathbb{C}_2$ represents a stronger set of constraints than $\mathbb{C}_1$. The simultaneous substitution is used for both weakening (when introducing new logic variables) and substitution (for unifying variables). When moving values down to an accessible world this substitution needs to be applied. Therefore accessibility is proof-relevant.

The pair $(w, \sqsubseteq)$ defines a Kripke frame and, because of reflexivity and transitivity of the accessibility relation, forms a Kripke model of the S_4 modal logic [Blackburn et al. 2001; Simpson 1994], with $w \longrightarrow \mathrm{Type}$ denoting propositions and satisfiability defined as function application $w \models A :\Leftrightarrow A\ w$. In the remainder we will use modal logic notation and terminology to structure our expositions, but otherwise not explore the logical interpretation further. We will explain all the necessary concepts as we go so readers need not be familiar with these concepts already. We will refer to objects in $w \longrightarrow \mathrm{Type}$ as being *modal types* or *Kripke-indexed types*.

Fig. 10 (middle) contains some basic constructions. Validity $\vdash A$ means that a computation of the given type A can be used without restriction in any world. We define a type for functions as point-wise functions of families. The box operator $\Box A$ denotes that a value (or computation) of type A can be used in any world accessible from a base world, i.e. at a node in the execution tree and all its descendants. A boxed value can be immediately used in the base world via the $\mathbf{T}$ combinator as defined in Fig. 10 (bottom). It uses the reflexivity of the accessibility relation given by the identity substitution and reflexivity of constraint entailment. The $\mathbf{4}$ combinator allows us to move a boxed value further down the execution tree without losing the box. It relies on the transitivity of accessibility which is implemented by composition of substitutions and transitivity of constraint entailment. The $\mathbf{K}$ combinator witnesses the semimonoidal (applicative [McBride and Paterson 2008] without pure) structure of the box operator $\Box$. It allows us to apply a boxed function to a boxed value without losing the box. We include it for completeness, but it is not used in the remainder. The combinators $\mathbf{K}$, $\mathbf{T}$ and $\mathbf{4}$ are the implementation of the S_4 axioms in our model.

$$
\begin{array}{l}
S_{\text{pure}}\ A \quad := \Box(A \to \mathbb{S}) \to \mathbb{S} \\
S_{\mathcal{S}}\ A \quad := \Box(A \to \text{env} \to \mathbb{S}) \to (\text{env} \to \mathbb{S}) \\
\hline
\eta \quad : \vdash A \to S_{\text{pure}}\ A := \lambda w\ a\ (k : \Box(A \to \mathbb{S})\ w).\mathrm{T}\ w\ k\ a \\
\gg\!= \quad : \vdash S_{\text{pure}}\ A \to \Box(A \to S_{\text{pure}}\ B) \to S_{\text{pure}}\ B := \\
\ \lambda w\ (m : S_{\text{pure}}\ A\ w)\ (f : \Box(A \to S_{\text{pure}}\ B)\ w)\ (k : \Box(B \to \mathbb{S})\ w). \\
\ \ m\ (\lambda w'\ (\omega : w \sqsubseteq w')\ (a : A\ w').f\ w'\ \omega\ a\ (4\ w\ k\ w'\ \omega)) \\
\text{Demonic} : \vdash S_{\text{pure}}\ V := \lambda w\ (k : \Box(V \to \mathbb{S})\ w). \\
\quad \textbf{let}\ \ell := \text{fresh}\ w\ \textbf{in} \\
\quad \textbf{let}\ w' := w, \ell\ \textbf{in} \\
\quad \textbf{let}\ \omega : w \sqsubseteq w' := \ldots\ \textbf{in} \\
\quad \forall \ell.k\ w'\ \omega\ \ell \\
\text{Assume} \ : \vdash \overline{\mathbb{F}} \to S_{\text{pure}}\ () := \lambda w\ \overline{\mathbb{F}}\ (k : \Box(() \to \mathbb{S})\ w). \\
\quad \textbf{match}\ \text{solver}\ w\ \overline{\mathbb{F}}\ \textbf{with} \\
\quad \mid \text{Some}\ (\overline{\ell \mapsto t}, \overline{\mathbb{F}'}) \Rightarrow \\
\quad\quad \textbf{let}\ w' := w[\overline{\ell \mapsto t}], \overline{\mathbb{F}'}\ \textbf{in} \\
\quad\quad \textbf{let}\ \omega : w \sqsubseteq w' := \ldots\ \textbf{in} \\
\quad\quad \overline{l \mapsto t} \to \overline{\mathbb{F}'} \to k\ w'\ \omega\ () \\
\quad \mid \text{None} \Rightarrow \top
\end{array}
$$

Fig. 11. Symbolic specification monad

We say that a type A is *persistent* if its values can always be used in all worlds accessible from a base, i.e. there is a designated function $\vdash A \to \Box A$. For first-order data, such as symbolic terms, persistence amounts to the definition of a substitution function. In general, function types $A \to B$ are not persistent, but boxed function types $\Box(A \to B)$ are, because all boxed types $\Box A$ are persistent via the 4 combinator. We denote the application of the persistence function for modal type A to a value $a : A\ w_0$ and accessibility witness $\omega : w_0 \sqsubseteq w_1$ as $a[\omega]$.

3.4 Symbolic Specification Monads

We define the backward predicate monad transformer S_{pure} on modal types in Fig. 11 and use it subsequently for the implementation of an interpreter. Similar to the shallow one W_{pure} from Section 2, it is defined as a continuation monad. The postcondition is wrapped in a $\Box$ since different execution branches, i.e. different worlds, will use it. Fig. 11 also defines some basic combinators for S_{pure}. For the purpose of presentation, we gray out the technical details around explicit handling of worlds and accessibility, and non-expert readers may choose to ignore these parts. η implements a return and $\gg\!=$ implements a bind operator for S_{pure}. Like for the postcondition in the definition of the monad, the continuation in the bind is wrapped in a $\Box$ [1]. To write monadic code we use the *do-notation*

$$[\omega]x \leftarrow m; k := m \gg\!= \lambda\ \_\ \omega\ x.\ k$$

that makes the world passing implicit but leaves the passing of the proof-relevant accessibility witness explicit.

For demonic non-deterministic choice of a term, we use a function $\text{fresh} : w \to \ell$ that picks a locally fresh name ℓ. We construct a new world w', which is w extended with ℓ, and also an accessibility witness $\omega : w \sqsubseteq w'$, for which we omit the details. Finally, we invoke the continuation

[1] The S_{pure} monad is not a strong monad, which prevents us from defining a bind with the usual type. However, it is an $\mathcal{L}$-strong monad [Kobayashi 1997] which ultimately results in a bind with the given type.

k in the world w' on ℓ (as a symbolic term). In the implementation of the assume guard we call the solver on the given formulas $\overline{\mathbb{F}}$. In the successful case, the simplification – unifications and residual formulas $\overline{\mathbb{F}'}$ – is used to construct the symbolic proposition. Furthermore, we need to construct a new world w' in which the unified variables are removed, the unifying substitution is applied to the old constraints, and the new residual formulas are added. In the failure case, the original formulas $\overline{\mathbb{F}}$ are inconsistent with the path condition. Consequently, we prune this execution path by emitting $\top$. The assert guard and angelic choice are implemented analogously.

Effects can be added to the pure specification monad by means of monad transformers. Fig. 11 shows the result S_δ of applying the state transformer with a symbolic store and uncurrying the result. The choice and guard operators can then be lifted to transformed monads like S_δ.

Equipped with these definitions, we can reimplement the interpreter of Section 2 in the symbolic specification monad. Where necessary, types have to be wrapped in $\Box$ and values and computations persisted, but otherwise the structure of the interpreter does not change. For instance, our combinator for pattern matching on sums becomes

$$\mathrm{Matchsum}\otimes : \vdash V \to \Box(V \to S_{\mathrm{pure}}\ A) \to \Box(V \to S_{\mathrm{pure}}\ A) := \lambda w\ V\ k_l\ k_r.$$
$$\left(\begin{array}{ll} [\omega_1]V_l & \leftarrow \mathrm{Demonic}; \\ [\omega_2]\_ & \leftarrow \mathrm{Assume}\ (V[\omega_1] = \mathrm{inl}\ V_l); \\ & T\ (k_l[\omega_1 \circ \omega_2]\ V_l[\omega_2]) \end{array}\right) \otimes \left(\begin{array}{ll} [\omega_1]V_r & \leftarrow \mathrm{Demonic}; \\ [\omega_2]\_ & \leftarrow \mathrm{Assume}\ (V[\omega_1] = \mathrm{inr}\ V_r); \\ & T\ (k_r[\omega_1 \circ \omega_2]\ V_r[\omega_2]) \end{array}\right)$$

and the main verification condition function

$$\begin{array}{l} \mathrm{VC}\ (f\ \overline{x} := e) : \mathbb{S}\ \varnothing := \mathbf{let}\ (\overline{\ell}, \delta, req, res, ens) := \mathrm{contract}\ f\ \mathbf{in} \\ \quad \mathbf{let}\ w : World := \overline{\ell}\ \mathbf{in} \\ \quad \mathbf{let}\ m : S_\delta\ ()\ w := \left\{\begin{array}{ll} [\omega_1]\_ & \leftarrow \mathrm{Assume}\ req; \\ [\omega_2]V & \leftarrow \mathrm{Exec}\ e; \\ & \mathrm{Assert}\ ens[\omega_1 \circ \omega_2, \ell \mapsto V]; \end{array}\right. \\ \quad \mathbf{in}\ \forall \overline{\ell}.m\ (\lambda w'\ \omega\ ()\ \delta'.\top)\ \delta \end{array}$$

3.5 Debug information

The debug production of $\mathbb{S}$ can be used to record information for any persistent type D during execution. For instance, this can include the path constraints that led to the executed branch and the program variable store at that point.

Our implementation in Section 6 will record such debug information automatically for all potential proof obligations, i.e. all assert ($\mathbb{F} \wedge \mathbb{S} \mid (l \mapsto t) \wedge \mathbb{S}$) and false ($\bot$) nodes. Furthermore, the user can request debug nodes explicitly via ghost commands in the programs and in the contracts or automatically for all calls of a particular function. We discuss examples of this debug information in Section 3.7 and in Section 4.1.

3.6 Postprocessing

To finalize the verification we can submit the generated verification conditions to an automated or alternatively to an interactive theorem prover. There are multiple cases that may necessitate human interaction. For instance, unsatisfiable verification conditions due to bugs in the program or the specification, or alternatively the absence of a sufficiently complete solver, because the application

$$
\begin{array}{l}
\forall(\mathrm{xs} : \mathrm{list\ int}), \mathrm{true} = \mathrm{true} \to \\
\quad (\mathrm{nil} = \mathrm{xs} \to \exists(sm : \mathrm{int} * \mathrm{int})(l : \mathrm{int}), (sm, l) = (0,0,0) \wedge \\
\qquad \exists(s\ m : \mathrm{int}), (s,m) = sm \wedge s \le m * l \wedge 0 \le l \wedge \top) \wedge \\
\quad (\forall(y : \mathrm{int})(ys : \mathrm{list\ int}), y :: ys = xs \to \exists(ys' : \mathrm{list\ int}), ys' = ys \wedge \mathrm{true} = \mathrm{true} \wedge \\
\qquad \forall(sml : \mathrm{int} * \mathrm{int} * \mathrm{int})(sm : \mathrm{int} * \mathrm{int})(l : \mathrm{int}), (sm, l) = sml \to \\
\qquad \forall(sm : \mathrm{int}), (s,m) = sm \to s \le m * l \to 0 \le l \to \\
\qquad \forall(sm' : \mathrm{int} * \mathrm{int})(l' : \mathrm{int}), (sm', l') = sml \to \forall(s'm' : \mathrm{int}), (s', m') = sm' \to \\
\qquad (m' < y \to \exists(sm'' : \mathrm{int} * \mathrm{int})(l'' : \mathrm{int}), (sm'', l'') = (s' + y, y, l' + 1) \wedge \\
\qquad\quad \exists(s''m'' : \mathrm{int}), (s'', m'') = sm'' \wedge s'' \le m'' * l'' \wedge 0 \le l'' \wedge \top) \wedge \\
\qquad (m' \ge y \to \exists(sm'' : \mathrm{int} * \mathrm{int})(l'' : \mathrm{int}), (sm'', l'') = (s' + y, m', l' + 1) \wedge \\
\qquad\quad \exists(s''m'' : \mathrm{int}), (s'', m'') = sm'' \wedge s'' \le m'' * l'' \wedge 0 \le l'' \wedge \top)).
\end{array}
$$

Fig. 12. The VC for summaxlen generated with the shallow executor.

specific theory is undecidable or because a solver cannot be integrated without enlarging the trusted code base. In any case, we want to make it as easy as possible for a user to pinpoint the source of a problem or prove VCs manually. To that end we implement a postprocessing phase that simplifies the output of the symbolic executor. By implementing this phase ourselves instead of relying on an automated solver, we can in particular pay attention to not disturb the control-flow structure encoded in the VC and more importantly that the recorded debug information stays consistent through this transformation.

In particular, we want to remove parts that correspond to explored execution paths with fully solved proof obligations, and to instantiate quantifiers by using the unifications provided by the solver. Consider the summaxlen example. At the node where the postcondition of the recursive call is assumed, the output of the symbolic executor contains a formula of the form

$$\forall \mathrm{sml\ sm\ l}.(\mathrm{sml} \mapsto (\mathrm{sm}, \mathrm{l})) \to \forall \mathrm{s\ m}.(\mathrm{sm} \mapsto (\mathrm{s}, \mathrm{m})) \to \ldots$$

which can be simplified to $(\forall \mathrm{l\ s\ m}. \ldots)$, essentially fusing two one-level pattern matches into a single multi-level one.

This can be implemented by the following transformation

$$(\forall \Sigma. \overline{\mathbb{F}} \to (l \mapsto t) \to \mathbb{S}) \rightsquigarrow (\forall (\Sigma - l). \overline{\mathbb{F}[l \mapsto t]} \to \mathbb{S}) \quad l \in \Sigma$$

which also allows for assumed formulas between the quantifier and the unification. Dually, we instantiate existentials with asserted formulas and unifications. Transformations such as

$$\forall \ell. \bot \rightsquigarrow \bot \qquad \exists \ell. \top \rightsquigarrow \top$$

are sound and complete for languages that have only inhabited types. Otherwise, the universal transformation is sound but can lead to incompleteness bugs, and the existential one is unsound.

The separation logic extension that we describe in Section 4 uses a heuristic that makes heavy use of angelic non-determinism to try different chunks of a symbolic heap. For this we found it helpful to be more aggressive and distribute existentials over the disjunction

$$\exists \Sigma. \overline{\mathbb{F}} \wedge (\mathbb{S}_1 \vee \mathbb{S}_2) \rightsquigarrow (\exists \Sigma. \overline{\mathbb{F}} \wedge \mathbb{S}_1) \vee (\exists \Sigma. \overline{\mathbb{F}} \wedge \mathbb{S}_1)$$

3.7 Example

Running our symbolic executor on the summaxlen example, with a solver that performs unification modulo the constructor theories of products and list, but without support for arithmetic, yields the

$$c ::= \mathcal{H}\ \overline{v} \qquad h ::= \overline{c} \qquad C ::= \mathcal{H}\ \overline{V} \qquad H ::= \overline{C}$$

Fig. 13. Shallow and deep heaps and chunks

following verification condition

$$\begin{aligned}&\forall(\mathrm{y} : \mathrm{int})\ (\mathrm{ys} : \mathrm{list\ int}).\quad \forall(\mathrm{l\ s\ m} : \mathrm{int}).\mathrm{s} \le \mathrm{m} * \mathrm{l} \to 0 \le \mathrm{l} \to \\ &\quad (\mathrm{m} < \mathrm{y} \to \mathrm{s} + \mathrm{y} \le \mathrm{y} * (\mathrm{l}+1) \wedge 0 \le \mathrm{l}+1 \wedge \top) \wedge \\ &\quad (\mathrm{m} \ge \mathrm{y} \to \mathrm{s} + \mathrm{y} \le \mathrm{m} * (\mathrm{l}+1) \wedge 0 \le \mathrm{l}+1 \wedge \top)\end{aligned}$$

In the nil case ($xs = []$), the postcondition $0 \le 0 * 0 \wedge 0 \le 0$ can be proved automatically by partial evaluation, or in this case even concrete evaluation. The part of the VC related to that branch has been fully removed. For the cons case ($\mathrm{xs} = \mathrm{y} :: \mathrm{ys}$) the logic variable xs of the contract – which is also the initial value of the program variable xs – is substituted by $\mathrm{y} :: \mathrm{ys}$ and the quantifier removed by the postprocessing phase. The second part of the first line of the VC contains the result of assuming the postcondition from the recursive call, simplified as described in Section 3.6. The last two lines contain obligations for proving the postcondition for both branches. The postprocessing phase removed translated pattern matches on products here as well. This VC can then be solved automatically by a solver with support for non-linear arithmetic.

Compare this to the VC generated with shallow execution in Fig. 12 which retains the propositional translation of the pattern matches and has not solved the nil case automatically. In particular, this clutter impairs a user to debug verification failures.

Asking for debug information using a ghost statement[2] on the last line of summaxlen will result in two debug nodes for both outcomes of the if-statement. The second of which contains the following information: the logic variables in scope are $\mathrm{y}, \mathrm{ys}, \mathrm{l}, \mathrm{s}, \mathrm{m}$, and the path constraints are $\mathrm{m} \ge \mathrm{y} \wedge 0 \le \mathrm{l} \wedge \mathrm{s} \le \mathrm{m} * \mathrm{l}$. Furthermore, the symbolic variable store contains the following mappings of program variables to symbolic terms:

$$\begin{array}{llll}\mathrm{xs} \mapsto \mathrm{y} :: \mathrm{ys} & \mathrm{ys} \mapsto \mathrm{ys} & \mathrm{s} \mapsto \mathrm{s} & \mathrm{sml} \mapsto ((\mathrm{s},\mathrm{m}),\mathrm{l}) \\ \mathrm{y} \mapsto \mathrm{y} & \mathrm{l} \mapsto \mathrm{l} & \mathrm{m} \mapsto \mathrm{m} & \mathrm{sm} \mapsto (\mathrm{s},\mathrm{m})\end{array}$$

In particular, in this case the mapping for the formal parameter xs shows the call pattern that together with the path constraints determines the execution path. For functions that assign new values to the formal parameter variables, the initial contents needs to be copied for the same effect.

4 SYMBOLIC HEAP SEPARATION LOGIC

Smallfoot [Berdine et al. 2005b] is a tool for checking program specifications that only describe the shape of pointer data structures rather than their contents. It works on a decidable fragment [Berdine et al. 2005a] of separation logic where all assertions are of the form $P \wedge Q$, where P is a pure proposition (the path condition) and Q is a separating conjunction of spatial heap predicates (the *symbolic heap*). In particular, the separating implication (magic wand) or septraction connective, which are generally used for spatial weakest precondition rules, is absent. Furthermore, Berdine et al. [Berdine et al. 2005c] present a symbolic execution method for automatically proving Hoare triples in this fragment.

Many verifiers [Distefano and Parkinson J 2008; Jacobs et al. 2010; Müller et al. 2016] adopted this approach, but also extended it to reason about the contents of data structures. In this section, we present how such a symbolic heap can be integrated into our method. Essentially, we extend

[2]The variable store in the debug information for proof obligations arising from the postcondition only contains the final state of the formal parameters of the function, since all other program variables are not in scope anymore.

$$\text{produce}_{\text{chunk}} : c \to \text{W}_{heap}\ () := \lambda c.\, h \leftarrow \text{get}_{\text{heap}};\ \text{put}_{\text{heap}}\ (c :: h)$$

$$\begin{aligned}\text{consume}_{\text{chunk}} : c \to \text{W}_{heap}\ () := \lambda\, c.\, (c_1..c_n) &\leftarrow \text{get}_{\text{heap}};\ i \leftarrow \mathit{angelic};\\ &\textbf{let } (\mathcal{H}\ \overline{v}) = c \textbf{ in let } (\mathcal{H}'\ \overline{v}') = c_i \textbf{ in}\\ &\textbf{if } \mathcal{H} = \mathcal{H}' \textbf{ then } \text{assert } (\overline{v} = \overline{v}') \textbf{ else fail};\\ &\text{put}_{\text{heap}}\ (c_1 \ldots c_{i-1}, c_{i+1} \ldots c_n)\end{aligned}$$

Fig. 14. Producing and consuming chunks

our shallow and symbolic specification monads with an additional piece of state: a shallow and symbolic separation logic heap, respectively:

$$W_{heap}\ x := (x \to \delta \to h \to \mathbb{P}) \to \delta \to h \to \mathbb{P}$$

$$S_{heap}\ x := \square(x \to \delta \to H \to \mathbb{S}) \to \delta \to H \to \mathbb{S}$$

These heaps have the syntax depicted in Fig. 13: a list of chunks, which themselves are abstract spatial predicate constructors applied to concrete or symbolic arguments. Similarly to pure predicates, we allow these spatial predicates to be custom-defined by the user and manipulated with user-provided lemmas, provided that both are backed up by implementations in the underlying Iris model. In this way, we can support arbitrary separation logic assertions, even though non-trivial manipulations of custom assertions require user annotations in the form of lemma invocations.

In terms of this additional state, we can then define $\text{produce}_{\text{chunk}}$ and $\text{consume}_{\text{chunk}}$ functions, the shallow versions of which are shown in Fig. 14. The naming of these functions stems from the resource interpretation of separation logic. $\text{produce}_{\text{chunk}}$ adds a single spatial predicate to the heap and $\text{consume}_{\text{chunk}}$ implements a heuristic for removing one by angelically choosing an element from the heap and asserting equality. Instead of the simple equality $c = c_i$, the shallow executor can do better and decide equality of the predicate names first, which never results in a blocked meta-computation since these are always specified concretely by the user. However, in general it cannot inspect the arguments without blocking. The symbolic version can improve upon this by looking at the arguments as well, and avoiding the backtracking semantics of the heuristic when the arguments already match a chunk on the heap exactly, and for precise predicates [O'Hearn et al. 2009] when only the "input" arguments match exactly.

Using chunks as a building block, we can define syntax for arbitrary structured assertions to be used in pre- and postconditions and more general produce and consume functions that interpret them in a specification monad. However, for space reasons we do not go into this further. The definitions are generally similar to those used in FVF [Jacobs et al. 2015] and μVeriFast [Devriese 2019].

4.1 Example: Linked lists

To give you an idea of practical use of our separation logic solver, we develop predicates and functions for dynamically heap-allocated singly linked lists [Reynolds 2000], implemented in our object language. We represent pointers ptr to the heap as integers, and linked lists llist as nullable pointers

$$\text{ptr} := \text{int} \qquad \text{llist} := \text{ptr} + ().$$

We implement linked lists in terms of heap-allocated pairs, consisting of a single element and a tail pointer, for which we use the following primitive procedures: mkcons to allocate a new pair on the heap, fst and snd to access the components of a heap-allocated pair, and setfst and setsnd to

$\forall x : \text{int}, q : \text{llist}.$	$\{\top\}$	$\text{mkcons}(x, q)$	$\{p.\ p \mapsto_p (x, q)\}$
$\forall p : \text{ptr}, x : \text{int}, q : \text{llist}.$	$\{p \mapsto_p (x, q)\}$	$\text{fst}(p)$	$\{r.\ r = x * p \mapsto_p (x, q)\}$
$\forall p : \text{ptr}, x : \text{int}, q : \text{llist}.$	$\{\exists y : \text{int}.\ p \mapsto_p (y, q)\}$	$\text{setfst}(p, x)$	$\{\_.\ p \mapsto_p (x, q)\}$
$\forall p : \text{ptr}, x : \text{int}, q : \text{llist}.$	$\{p \mapsto_p (x, q)\}$	$\text{snd}(p)$	$\{r.\ r = q * p \mapsto_p (x, q)\}$
$\forall p : \text{ptr}, x : \text{int}, q : \text{llist}.$	$\{\exists q' : \text{llist}.\ p \mapsto_p (x, q')\}$	$\text{setsnd}(p, q)$	$\{\_.\ p \mapsto_p (x, q)\}$

Fig. 15. Contracts for linked lists primitives

	$\{\top\}$	open_nil	$\{\text{inr}\ () \mapsto_l []\}$
$\forall z : (), xs : \text{list int}.$	$\{\text{inr}\ z \mapsto_l xs\}$	close_nil	$\{z = () * xs = []\}$
$\forall p : \text{ptr}, xs : \text{list int}.$	$\{\text{inl}\ p \mapsto_l xs\}$	$\text{open\_cons}(p)$	$\left\{\begin{array}{l}\text{match } xs \text{ with} \\ \mid\ [] \mapsto \bot \\ \mid\ y :: ys \mapsto \exists n : \text{llist}.\ p \mapsto_p (y, n) * \\ \qquad\qquad n \mapsto_l ys\end{array}\right\}$
$\forall \begin{array}{l} p : \text{ptr}, x : \text{int}, \\ xs : \text{list int}, n : \text{llist}\end{array}$	$\left\{\begin{array}{r} p \mapsto_p (x, n) * \\ n \mapsto_l xs \end{array}\right\}$	$\text{close\_cons}(p)$	$\{\text{inl}\ p \mapsto_l x :: xs\}$

Fig. 16. Lemmas about linked list heap predicates

update the components. The signatures and contracts of these primitives are shown in Fig. 15. In our implementation (Section 6), we instantiate the memory of the Iris model as being a finite map of pointers to pairs:

$$\text{mem} := \text{ptr} \to (\text{int}, \text{llist})$$

and implement the primitives directly in Coq. For the separation logic contracts we use of two spatial points-to predicates, one for a single heap-allocated pair and one for a heap-allocated list:

$$\begin{array}{ll} p \mapsto_p (x, q) & p : \text{ptr}, x : \text{int}, q : \text{llist} \\ q \mapsto_l xs & q : \text{llist}, xs : \text{list int} \end{array}$$

The definition of these predicates is part of the model with the list predicate defined recursively in terms of the pair predicate [Reynolds 2000]:

$$\begin{array}{lcl} q \mapsto_l [] & := & q = \text{inr}\ () \\ q \mapsto_l (x :: xs) & := & \exists (p : \text{ptr})(n : \text{llist}), (q = \text{inl}\ p) * (p \mapsto_p (x, n)) * (n \mapsto_l xs) \end{array}$$

We do not expose the recursive definition to the VCGs. Instead, we declare and use lemmas for the folding/unfolding of the recursion which are defined in Fig. 16. Such lemmas can generally be inserted to aid the verifier make non-trivial reasoning steps and need to be proven sound by the user in the underlying model. Currently, we require the user to give hints to the VCGs when a lemma needs to be used to transform the proof state, which the user provides through ghost statements in the code. In the future we are planning to support user-provided heuristics which automatically invoke such lemmas where needed.

Fig. 17 shows the two functions append and $\text{append}_{\text{loop}}$ with their contracts. Together, they implement an in-place append for linked lists. $\text{append}_{\text{loop}}$ follows the chain of the first linked list till the end and updates the (null) tail pointer to point to the second list instead. At each step, it unfolds one-level of the recursive list predicate using *open_cons* to reveal the cons cell. After returning from a recursive call the predicate needs to be folded again using *close_cons*. Spatial predicates for the empty list, which hence do not contain any cons cells, can be discarded using close_nil.

```
{p ↦_l xs * q ↦_l ys}
append(p : llist, q : llist) : llist :=
  case p of inl n ⇒ call append_loop n q; p
            inr z ⇒ lemma close_nil z; q
{r. r ↦_l xs ++ ys}
```

```
{inl p ↦_l xs * q ↦_l ys}
append_loop(p : ptr, q : llist) : () :=
  lemma open_cons p;
  let t := foreign snd p in
  case t of inl n ⇒ call append_loop n q
            inr z ⇒ lemma close_nil z;
                    foreign setsnd p q;
  lemma close_cons p
{_. p ↦_l xs ++ ys}
```

Fig. 17. Appending two linked lists

Omission of any of the lemma ghost statements in the code will result in a verification failure: the consumption of some chunk will fail if nothing suitable is found on the heap. For instance, removing the first ghost lemma statement that invoked open_cons in $\text{append}_{\text{loop}}$ will the result in the following failure for the foreign call to snd for which we can again record debug information.

The chunk to be consumed is

$$p \mapsto_p (x', q')$$

which comes from the precondition of snd with x' and q' being existential variables. However, the heap only contains the chunks incompatible chunks

$$\text{inl}\ p \mapsto_l xs \qquad q \mapsto_l ys$$

and hence execution fails. We can of course record the state again. The local variable store will still contain its initial mapping

$$\text{p} \mapsto \text{p} \qquad \text{q} \mapsto \text{q}$$

since no unifications have been performed yet.

4.2 Discussion

The separation logic primitives of the shallow W_{heap} monad defined in this section are not representative. In particular, the backtracking heuristic of $\text{consume}_{\text{chunk}}$ produces many unnecessary branches, and the reasoning steps performed by explicit user-provided ghost statements are usually performed as part of a proof script or are even automatic by evaluation or delta expansion.

For our purposes, the shallow VC serves primarily as a means to factorize the soundness proof of the symbolic VC for which it defines the *allowed behavior*. More realistically, a shallow VC would be implemented using a specification monad (e.g. W_{pure}) defined on top of proper separation logic propositions.

Using such a monad, it is easy to see that $\text{produce}_{\text{chunk}}$/$\text{consume}_{\text{chunk}}$ are the spatial equivalents of assume/assert. Indeed, given an interpretation $[\![\_]\!]$ of chunks as separation logic propositions, we can use the definitions:

$$\text{produce}_{\text{chunk}}\ c : W_{\text{pure}}\ () := \lambda\ k.\ [\![c]\!] \mathrel{-\!\!*} k\ ()$$
$$\text{consume}_{\text{chunk}}\ c : W_{\text{pure}}\ () := \lambda\ k.\ [\![c]\!] * k\ ()$$

$$\mathcal{R}_{\lesssim}[\![A, a]\!] \subseteq \{(w, \iota_w, A, a)\}$$
$$\mathcal{R}_{\lesssim}[\![\mathrm{V}, \mathrm{v}]\!] = \{(w, \iota_w, V, v) \mid v = V[\iota_w]\}$$
$$\mathcal{R}_{\lesssim}[\![\mathbb{S}, \mathbb{P}]\!] = \{(w, \iota, \mathbb{S}, \mathbb{P}) \mid (\iota \models \mathbb{S}) \Rightarrow \mathbb{P}\}$$
$$\mathcal{R}_{\lesssim}[\![\Box A, a]\!] = \{(w, \iota, A, a) \mid \forall w', \omega : w \sqsubseteq w', \iota'.\, \iota = \iota' \circ \omega \rightarrow (w', \iota', A, a) \in \mathcal{R}_{\lesssim}[\![A, a]\!]\}$$
$$\mathcal{R}_{\lesssim}[\![A \rightarrow B, a \rightarrow b]\!] = \{(w, \iota, f_s, f_c) \mid \forall (w, \iota, v_s, v_c) \in \mathcal{R}_{\lesssim}[\![A, a]\!].\, (w, \iota, f_s\ v_s, f_c\ v_c) \in \mathcal{R}_{\lesssim}[\![B, b]\!]\}$$

Fig. 18. Refinement relation to prove soundness of symbolic execution – selected rules

5 REFINEMENT

The two executors that we described in Sections 2 and 3 are essentially the same program albeit implemented in two different *languages*. The question arises in what way these two implementations are equivalent, and how we can establish that fact formally. Ultimately, the property we want for their outputs is dictated by our soundness requirement, which is expressed by the following lemma.

Lemma 5.1 (Soundness of symbolic execution). *Given a program, if the symbolic verification condition holds for a function, then so does the shallow one, i.e.*

$$\forall f, (\epsilon \models VC\ f) \Rightarrow vc\ f.$$

We have to generalize this in two ways. First, we need to consider other moments of the execution, instead of just the final closed result. That means we are in a world w and want to consider the implication $(\iota_w \models \mathbb{S}) \rightarrow \mathbb{P}$ for valuations ι_w of that world:

$$\iota_{(\Sigma,\mathbb{C})} \subseteq \{\iota_\Sigma \mid \iota_\Sigma \models \mathbb{C}\}$$

Second, we need to generalize this to other types. We define this by means of a logical relation [Tait 1967] $\mathcal{R}_{\lesssim}$ that we discuss shortly. In particular, we can relate predicate transformer monads, e.g. S_{pure} and W_{pure}. For these types $\mathcal{R}_{\lesssim}$ encodes a notion of refinement [Back and Wright 2012; Morgan 1990].

The refinement relation is defined in Fig. 18. $\mathcal{R}_{\lesssim}[\![A, a]\!]$ relates symbolic computations of type A with pure computations of type a at a given world w and valuation ι_w. For propositions and formulas, the relation is as just discussed. For first-order data like symbolic terms, stores, heaps, etc. it is equality after instantiation. As usual, related functions map related inputs to related outputs. The most interesting case is that of a boxed typed $\Box A$. It requires that in every accessible world $\omega : w \sqsubseteq w'$, the symbolic computation is related to the pure one. However, we also need to consider valuations in the new world, we require relatedness for every valuation $\iota_{w'}$ that is compatible with (that extends) the old one, which is expressed by composition with the substitution that witnesses the accessibility.

Proof sketch of Lemma 5.1. Unfolding the definition of the logical refinement relation on propositions $\mathcal{R}_{\lesssim}[\![\mathbb{S}, \mathbb{P}]\!]$, we can see that the soundness statement is equivalent to the inclusion in the relation in the empty (initial) world:

$$(\varnothing, \epsilon, VC\ f, vc\ f) \in \mathcal{R}_{\lesssim}[\![\mathbb{S}, \mathbb{P}]\!]$$

To prove that inclusion, we show that all constituent functions and monadic operators are related, e.g.

$$(w, \iota_w, \mathrm{Exec}\ e, \mathrm{exec}\ e) \in \mathcal{R}_{\lesssim}[\![M\ V, W\ v]\!]$$
$$(w, \iota_w, \gg=, \gg=) \in \mathcal{R}_{\lesssim}[\![S\ A \rightarrow \Box(A \rightarrow S\ B) \rightarrow S\ B, \ldots]\!]$$

This is mostly mechanical, since the two implementation have the same structure. The only meaningful difference is in the implementation of the *Assume* and *Assert* commands that call the solver, e.g.

$$(w, \iota_w, \text{Assume}, \text{assume}) \in \mathcal{R}_{\lesssim}[\![\mathbb{F} \to S_{\text{pure}}\ (), \mathbb{P} \to W_{\text{pure}}\ ()]\!]$$

This property is established by reducing it to the correctness of the solver. □

Example: demonic choice. As an example of a relatedness proof, consider the demonic choice combinators (shown in Figures 2 and 11). After inlining the definitions, applying them to two related postconditions

$$(w, \iota_w, \text{Post}, \text{post}) \in \mathcal{R}_{\lesssim}[\![\Box(t \to \mathbb{S}), v \to \mathbb{P}]\!]$$

and using $\ell = \textbf{fresh}\ w$ the logical relation becomes

$$(\iota_w \models \forall \ell.\text{Post}\ (w, \ell)\ \omega\ \ell) \to \forall v.\text{post}\ v$$

where $\omega : w \sqsubseteq (w, \ell)$. After introducing the quantified value v on the right and instantiating the left quantifier it simplifies further to

$$((\iota_w, \ell \mapsto v) \models \text{Post}\ (w, \ell)\ \omega\ \ell) \to \text{post}\ v$$

Using relatedness of the postconditions it remains to show

$$((w, \ell), (\iota_w, \ell \mapsto v), \ell, v) \in \mathcal{R}_{\lesssim}[\![\mathrm{V}, \mathrm{v}]\!],$$

i.e. that the logic variable ℓ is related to v in (w, ℓ), which is immediate.

Discussion. An alternative approach would be to prove the soundness of the symbolic *VC* against the program logic directly. As briefly discussed at the end of Sec. 2.4 this proceeds in two steps: first show the monotonicity of the predicate transformers and then their soundness.

The monotonicity statement for symbolic predicate transformers $T : \Box(A \to \mathbb{S}) \to \mathbb{S}$ has to be properly generalized to account for world changes. In particular, we have to generalize the order relation on predicates $P, Q : \Box(A \to \mathbb{S})$ to account for using them in two different, but accessible worlds. More precisely, in a base world w_0 we may want to use P in w_1 with $\omega_1 : w_0 \sqsubseteq w_1$ and Q in w_2 with $\omega_2 : w_1 \sqsubseteq w_2$ and apply them to different but related values $a_1 : A\ w_1$ and $a_2 : A\ w_2$. Consequently, defining the monotonicity of predicate transformers precisely seems to require another custom-build logical relation. We attempted a direct proof using these ideas, which we ultimately gave up on, but we remain positive that the discussed generalization is sufficient to carry out this proof.

Factorizing the proof through the shallow *vc* also splits up the monotonicity proof. Specifically, the universal quantification over the witness in the logical relation for boxed types in Fig.18 takes care of the world change, and the remaining monotonicity proof for shallow predicate transformers can use *equal* values of a result type A.

6 KATAMARAN

Katamaran is a verifier for Sail [Armstrong et al. 2019] that implements the techniques described in this paper. Sail is a domain-specific language for executable specifications of instruction set architectures and Katamaran implements a variant of Sail called μSail, which is deeply embedded in the Coq theorem prover. Sail and μSail feature structured types such as lists, enums, records unions and bit-vectors[3]. The goal of Katamaran is to support semi-automatic proofs of ISA security properties, something we will report in more detail elsewhere.

[3]Katamaran is still in early development and does not yet support proof automation for bit-vectors.

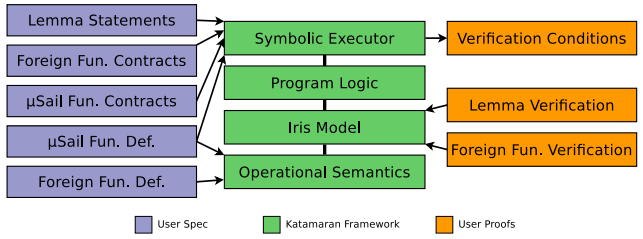

Fig. 19. Structure of Katamaran

The structure of Katamaran is depicted in Fig. 19. The semantics of µSail is defined both in terms of a small-step operational semantics and an axiomatic program logic interface based on separation logic Hoare triples. Specifications of programs consist of triples for all declared functions.

The framework is abstracted over an underlying separation logic, and any logic implementing the defined interfaces can serve as a model. The library comes with a pre-defined model that uses the Iris separation logic framework [Jung et al. 2018] together with an adequacy theorem that links the Iris model with the operational semantic.

It treats registers (global variables) and machine memory as resources. A user-provided runtime system defines what constitutes a machine's memory and provides access to it via foreign functions, i.e. functions callable from µSail but implemented in Coq. On top, we have a symbolic VC gen that is proved sound w.r.t. a shallow one, which in turn is proved sound to the program logic. For the symbolic executor, Katamaran includes a generic solver that augments a user provided one. It implements among other things unification modulo the constructor theory for structured datatypes using datatype generic proving and programming techniques [Altenkirch and Mcbride 2003; Hinze 2000].

As outlined in Section 4, µSail allows the invocation of lemmas (sometimes referred to as ghost statements), which instruct the verifier to take a proof step, which is currently the only form of non-trivial spatial reasoning in the VC gens. The user has to prove these lemmas in the underlying model, which can be done using Iris Proof Mode [Krebbers et al. 2018] for the included model.

The implementation uses an intrinsically-typed representation [Altenkirch and Reus 1999; Bach Poulsen et al. 2017; Benton et al. 2012], which means that all values, variables, expressions, statements, symbolic terms, stores, predicates etc. are constrained to be well-scoped and well-typed and all operations on them are type-preserving by construction. In particular, the generic solver, the symbolic executor and the program logic can ignore cases that are impossible due to typing. Dodds and Appel [Dodds and Appel 2013] report on similar benefits of integrating a typechecker in a verifier. Katamaran makes heavy use of the Equations [Sozeau and Mangin 2019] package for dependent pattern matching on typed terms.

The proof obligations for the user are the verification conditions, the lemmas used in ghost statements, and the verification of the foreign functions. These proofs are inputs to the various soundness theorems of the framework.

6.1 Iris model

We have developed a small-step operational semantics for µSail which a stepping relation of the form

$$(\gamma, \mu, \delta, e) \longrightarrow (\gamma', \mu', \delta', e')$$

with global state for registers γ, γ' and memory μ, μ', local variable store δ, δ' and expressions e, e'. We instantiated the Iris separation logic framework [Jung et al. 2018] with our operational

	Katamaran						Bedr.	VST	SLF
	Shallow VC		Symbolic VC			Solver			
	Branches	Pruned	Branches	Pruned	Time	Time	Time	Time	Time
append	4	0	2	0	0.0090	–	31.5	2.61	–
$\text{append}_{\text{loop}}$	34	26	3	1	0.0667	–			0.99
copy	33	15	3	1	0.0209	–	–	–	0.95
length	5	3	3	1	0.0469	0.15	16.8	–	0.78
reverse	2	0	1	0	0.0051	–	20.0	2.34	–
$\text{reverse}_{\text{loop}}$	24	14	3	1	0.0243	0.25			–
summaxlen	3	0	3	0	0.1729	–	–	–	–
Lemmas					0.3304		1.05	–	0.33

Table 1. Linked list comparison.

semantics to obtain a model for our axiomatic program logic and proved that all rules of our program logic are admissible in the model. This contains a catch-22: the program logic and executors are parameterized over a set of given contracts and thus the verification conditions express the validity of a contract for a function body under the assumption that all contracts hold for recursive calls. The guarded recursion machinery of Iris provides the means to tie this knot. Moreover, Iris provides the necessary adequacy lemmas to connect the specified contracts to the operational semantics for which we state a special case for functions with pure contracts that can be expressed without referring to the program logic or another abstraction. This demonstrates that the implementation, soundness proofs and Iris model of Katamaran can be combined to obtain end-to-end results stated purely in terms of the operational semantics of the language.

Lemma 6.1 (Adequacy of pure contract). *Assuming the generated verification conditions hold for all functions, then for any function f with a pure (non-spatial) contract $(\Sigma, \overline{V}, req, res, ens)$ the following holds*

$$\forall \iota_\Sigma\ \delta\ \delta'\ \gamma\ \gamma'\ \mu\ \mu'\ v.\ req\ \iota_\Sigma \Rightarrow (\gamma, \mu, \delta, call\ f\ \overline{V[\iota_\Sigma]}) \longrightarrow (\gamma', \mu', \delta', v) \Rightarrow ens\ (\iota_\Sigma, res \mapsto v)$$

6.2 Singly linked lists

An important contribution of this paper is a principled way to support simplifying path constraints and pruning unreachable paths. Table 1 illustrates this by showing the number of execution paths explored in shallow and symbolic VC generation for the linked list examples of Section 4. The table shows the amount of execution branches explored by both, the shallow and the symbolic executor, and the number of branches that have been pruned early, i.e. executions that did not reach the end of a function or more precisely the end of the postcondition of a function. For the listed functions, the pruning of the branches in the symbolic executor comes solely from the $\bot$ assertion in the postcondition of the *open_cons* lemma.

As previously explained, the simplification of constraints is crucial for preventing path explosion in a practically viable symbolic execution tools. In particular in these examples, the eager unification of variables helps the symbolic execution, because it helps the heuristics of deterministically instead of angelically selecting a chunk from the symbolic heap that we briefly described in Sec. 4. In fact, all consumed chunks in the examples are selected deterministically by the symbolic executor and angelically by the shallow one. The table shows that simplifying path conditions and pruning

```
{(∃c. pc ↦ c ∗ 𝒱(c)) ∗ (∀r ∈ GPR. ∃w. r ↦ w ∗ 𝒱(w))}
store(rs : GPR, rb : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap rb in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let w := call read_reg rs in
  lemma move_cursor base_cap c;
  call write_mem c w;
  call update_pc;
  true
{(∃c. pc ↦ c ∗ 𝒱(c)) ∗ (∀r ∈ GPR. ∃w. r ↦ w ∗ 𝒱(w))}
```

Fig. 20. Capability safety for the store instruction.

unreachable paths in the symbolic VCG strongly reduces the number of execution paths compared to the shallow VCG, even in these relatively simple examples. While solving these shallow VCs is still tractable, it quickly becomes untractable for larger examples.

Table 1 also shows a comparison of total verification time in seconds for the example functions of our symbolic execution and similar functions in related work. The Bedrock framework [Chlipala 2011] works with a custom low-level imperative language, and the Separation Logic Foundations [Charguéraud 2020] on custom ML-like language. The Verified Software Toolchain [Cao et al. 2018] operates on C programs. All three of these systems use meta-programming for reasoning about a shallowly-embedded logic.

While the symbolic executor is in principle extractable we have not done so. The numbers show the time to generate the VC by evaluation in Coq's typechecker and subsequent proving of the VC. For the length and $\text{reverse}_{\text{loop}}$ functions we needed to define pure predicates that encode the functional correctness specification and the Solver column shows the time to prove a user-defined solver for these predicates correct. The other systems have to perform similar reasoning steps during their verifications.

The measurements were taken using Coq 8.15.1 on an AMD Ryzen 9 5950X with the exception of the Bedrock column, for which we used Coq 8.4.6. For all the linked list functions simplification and postprocessing resulted in a trivial VC. Hence, these verification problems are fully solved by computational reflection [Barendregt and Barendsen 2002; Beeson 2016; Boutin 1997] which is more performant than proof term construction. As a result Katamaran is more than an order of magnitude faster. For the summaxlen, the generated VC is the one we discussed in Sec. 3.7.

6.3 MinimalCaps

We have instantiated our approach in our capability machine case study, called MinimalCaps [Anonymized 2021], for which we prove that the capability safety property [Devriese et al. 2016; Georges et al. 2021; Swasey et al. 2017; Van Strydonck et al. 2019] holds. The MinimalCaps case study exemplifies the intended usage of Katamaran, the verification of security guarantees offered by ISAs. However, we limit the discussion of our case study to the evaluation of Katamaran in verifying that the capability safety property holds.

In the case study we have defined contracts for each available instruction, as well as the fetch-decode-execute loop. The contract for the store instruction is shown in Fig. 20 as a separation logic Hoare triple around the definition of the instruction. The meaning of the shown pre- and postcondition can be summarized as: if we start with a capability safe configuration, we will end (upon successful execution) with a capability safe configuration. In other words, the instructions of the MinimalCaps case study cannot break the capability safety property. The store instruction has 3 parameters, a register containing the value to be stored in memory, a register containing a capability as a basis for the memory location and an immediate value that will be added to the cursor of the capability to get the exact location for the memory store (which needs to be within the bounds of the capability). The body of the store instruction consists of 4 let bindings that will read the argument registers and derive a capability based on the one in the *rb* register. Furthermore, we have a lemma invocation to aid Katamaran in the verification of this construct and some function calls to perform the actual write to memory and increment the program counter. The boolean at the end of the store definition is used to indicate that the fetch-decode-execute loop should continue looping.

The verification of the store contract starts with the precondition stating that all accessible registers and memory locations contain safe values. Throughout the let bindings Katamaran will learn more about the contents of some of these registers. For example, by invoking read_reg_cap, we learn that the register *rb* has to contain a capability (the contract for read_reg_cap states this, otherwise the machine will go into a failed state). The binding for *w* corresponds to the contents of register *rs*, and from the precondition we know that this value will be safe. The only missing piece of information that Katamaran still needs to verify this contract is that the capability to perform the write with, i.e., *c*, needs to be safe. This information is learned with the lemma move_cursor, which will derive the safety of a capability that is derived from a safe capability. At the end of the store instruction, Katamaran will be able to verify that the postcondition holds.

The MinimalCaps case study consists 444 LoC for 48 μSail functions and 12 LoC for 3 foreign functions. We define 8 lemmas to guide the spatial reasoning which are used in a total of 34 invocations. Moreover, we define one pure predicate for a simple permission lattice and include a solver for it. The interesting and complicated part of the development is the verification of the lemmas and safety of the foreign functions that provide access to the machine's memory. This verification is done directly in the Iris model using Iris Proof Mode. The functions implemented in μSail like the store instruction in Fig. 20 are much simpler to verify but represent the majority of the code. In fact, a large subset does not even touch memory. Katamaran's goal is to automate this *boring* bulk of security property verification.

The time to verify all 48 μSail functions with the symbolic executor is 0.5356s, meaning it is fast enough to allow us to interactively experiment with definitions in our case study, continously expand it and immediately verify the corresponding contracts. In total the symbolic executor explores 117 execution paths of which it prunes 17 early, while the shallow executor explores 57867 paths with 42686 pruned paths.

7 RELATED WORK

The literature on the topics we touch in this paper is vast. For the discussion, we focus on related work with a high degree of assurance.

Certified verifications. The ecosystem of verifiers, solvers, and theorem provers is evolving to the extent that we can formally establish program verification results with machine checked proofs. One approach, the one promoted in this paper, is to verify the implementation of a verifier itself, so

that we can automagically trust each run. This is sometimes called the *autarkic style* [Barendregt and Barendsen 2002; Beeson 2016] or *proof by computational reflection* [Boutin 1997].

Existing work has already mechanized various subsets of the pipeline of verifiers. For instance, [Vogels et al. 2009] contains a mechanized formalization of an intermediate verification languages (IVL), together with a proof of soundness of its VCgen, and [Vogels et al. 2010] goes even further to mechanize an algorithm for compact VCs [Flanagan and Saxe 2001; Leino 2005]. To boost confidence in tools, we would ideally integrate verified implementations in their code. Unfortunately, this is not common practice, since mechanized implementations are usually not built for performance. But it is also not unheard of. For instance, [Appel 2011] report competitive performance of an extracted version of VeriSmall versus the original Smallfoot implementation.

A more common approach is to verify individual runs of a verifier, the *skeptical style*. For instance, we can instrument the implementation of a verifier [Parthasarathy et al. 2021] to produce a certificate that can then be checked in a theorem prover, to mechanically verify that it indeed witnesses a valid derivation in the program logic. A second variation, is to autarkically verify a checker that validates certificates, but we are not aware of such a system for the purpose of program verification. Another option, is to integrate the tool with a theorem prover, by embedding the program logic in the logic of the prover and interacting with it to build a proof term for the derivation, by interactive, semi-automatic, or mostly-automated [Cao et al. 2018; Chlipala 2011; Krebbers et al. 2018; Tuerk 2009] means. This approach has been used support large subsets of realistic programming languages such as C [Cao et al. 2018] or pure Caml [Charguéraud 2010].

Specification and Dijkstra monads. As we have shown in this paper, there are clearly variations possible in the definition of specification monads. We believe that other variations are possible, such as using the propositions of other domain-specific logics, be they shallowly- or deeply-embedded. VeriSmall [Appel 2011] also uses a continuation monad to model non-determinism, but since it works on a decidable fragment of separation logic, it can even use a boolean result type. Featherweight VeriFast (FVF) [Jacobs et al. 2015] uses an intensional specification monad that supports angelic and demonic choice, which is not based on continuation monads.

Dijkstra monads [Ahman et al. 2017; Jacobs 2014; Maillard et al. 2019; Swamy et al. 2013] can be constructed by indexing a computation monad with a specification monad, to co-design programs and specifications, and verify correctness. We believe we could similarly index the specification monad S with the monad W, and implement the symbolic executor with the concrete one as the specification. This would essentially fuse the implementation of the symbolic executor with the refinement proofs of Section 5.

Verified symbolic execution. Although other tools do not make them as explicit as we do, Kripke frames naturally arise in the implementation of symbolic executors and can clearly be seen in the proofs. For instance, VeriSmall [Appel 2011] represents variables as numbers and keeps track of an upper bound of used variables to allocate fresh ones. The upper bound with the path constraints form the worlds and the proof irrelevant accessibility is the inequality $\leq$ and constraint entailment. Similarly, FVF [Jacobs et al. 2015] uses sets of variables and proof irrelevant set inclusion.

The main soundness theorem of VeriSmall uses an intricate induction scheme, that involves the treatment of fresh variables including a number ghost that separates numbers representing program variables from logic variables. In our mechanization, we only used standard structural induction. We believe that the difficulty in VeriSmall arises during induction over assertions or statements that contain logic variables for which we needed to add another $\Box$-operator to account for the fact that the world can change during traversal. Since accessibility in VeriSmall is proof irrelevant, this additional $\Box$ would be invisible in the implementation of the symbolic executor, but in our experience needs to be accounted for in the proof.

Our approach draws many inspirations from FVF, but also improves on the method and systematizes it in several ways. FVF defines an approximation relation which is roughly equivalent to our logic relation for the specification monad types. However, FVF does not generalize this to other types, but defines the needed soundness statement adhoc as needed. Our logical relation systematically calculates the soundness theorem for every type. While FVF works with an intensional specification monad, the soundness theorem of FVF applies a CPS on top. When trying to use their specification monad without CPS we found that we needed additional proof obligations that encode a notion of stability: postcondition continue to hold under accessibility. Furthermore, FVF does not prove a general refinement relation for the monadic bind operator, but instead relies on the monad laws to reassociate operations and then use a CPS transformed refinement on the first operation.

REFERENCES

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3009837.3009878

Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, et al. 2014. The KeY platform for verification and analysis of Java programs. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer.

Thorsten Altenkirch and Conor Mcbride. 2003. *Generic Programming within Dependently Typed Programming*. Springer US, Boston, MA. https://doi.org/10.1007/978-0-387-35672-3_1

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic (LNCS, Vol. 1683)*, Jörg Flum and Mario Rodriguez-Artalejo (Eds.). Springer, 453–468.

Anonymized. 2021. *MinimalCaps Case Study*. https://github.com/katamaran-project/minimalcaps

Andrew W. Appel. 2011. VeriSmall: Verified Smallfoot Shape Analysis. In *Certified Programs and Proofs*. Springer Berlin Heidelberg.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290384

Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz. 2001. Chapter 8 - Unification Theory. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). North-Holland, Amsterdam. https://doi.org/10.1016/B978-044450813-3/50010-2

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (dec 2017). https://doi.org/10.1145/3158104

Ralph-Johan Back and Joakim Wright. 2012. *Refinement calculus: a systematic introduction*. Springer Science & Business Media.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018). https://doi.org/10.1145/3182657

Henk Barendregt and Erik Barendsen. 2002. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning* 28, 3 (01 Apr 2002). https://doi.org/10.1023/A:1015761529444

Michael Beeson. 2016. Mixing Computations and Proofs. *Journal of Formalized Reasoning* 9, 1 (2016). https://doi.org/10.6092/issn.1972-5787/4552

Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning* 49, 2 (2012). https://doi.org/10.1007/s10817-011-9219-0

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. A Decidable Fragment of Separation Logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, Kamal Lodaya and Meena Mahajan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005b. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11804192_6

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005c. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*. Springer Berlin Heidelberg.

Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal Logic*. Cambridge University Press. https://doi.org/10.1017/CBO9781107050884

Samuel Boutin. 1997. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, Martín Abadi and Takayasu Ito (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 209–224.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1 (2018). https://doi.org/10.1007/s10817-018-9457-5

Arthur Charguéraud. 2010. Program Verification through Characteristic Formulae. *SIGPLAN Not.* 45, 9 (sep 2010). https://doi.org/10.1145/1932681.1863590

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. *SIGPLAN Not.* 46, 9 (sep 2011), 418–430. https://doi.org/10.1145/2034574.2034828

Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (2020). https://doi.org/10.1145/3408998

Adam Chlipala. 2011. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. *SIGPLAN Not.* 46, 6 (2011). https://doi.org/10.1145/1993316.1993526

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective Interactive Proofs for Higher-Order Imperative Programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/1596550.1596565

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer.

Dominique Devriese. 2019. Modular Effects in Haskell through Effect Polymorphism and Explicit Dictionary Applications: A New Approach and the μVeriFast Verifier as a Case Study *(Haskell 2019)*. ACM. https://doi.org/10.1145/3331545.3342589

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 147–162. https://doi.org/10.1109/EuroSP.2016.22

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975). https://doi.org/10.1145/360933.360975

Dino Distefano and Matthew J. Parkinson J. 2008. JStar: Towards Practical Verification for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1449764.1449782

Josiah Dodds and Andrew W. Appel. 2013. Mostly Sound Type System Improves a Foundational Program Verifier. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham.

Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–177.

Cormac Flanagan and James B. Saxe. 2001. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/360204.360220

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities. *Proc. ACM Program. Lang.* 5, POPL, Article 6 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434287

Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. 2009. Proving That Non-Blocking Algorithms Don't Block. *ACM SIGPLAN Notices* 44, 1 (Jan. 2009), 16–28. https://doi.org/10.1145/1594834.1480886

RDA Hendriks and Vincent van Oostrom. 2003. Adbmal. *Lecture Notes in Computer Science* 2741 (2003), 136–150.

Ralf Hinze. 2000. Generic programs and proofs. (2000).

Ralf Hinze. 2012. Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick. In *Mathematics of Program Construction*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 324–362.

R.John Muir Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* 22, 3 (1986). https://doi.org/10.1016/0020-0190(86)90059-1

Graham Hutton, Mauro Jaskelioff, and Andy Gill. 2010. Factorising folds for faster functions. *Journal of Functional Programming* 20, 3-4 (2010), 353–373. https://doi.org/10.1017/S0956796810000122

Bart Jacobs. 2014. Dijkstra Monads in Monadic Computation. In *Coalgebraic Methods in Computer Science*, Marcello M. Bonsangue (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg.

Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 6461. Springer Berlin Heidelberg.

Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* Volume 11, Issue 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Form. Asp. Comput.* 27, 3 (2015). https://doi.org/10.1007/s00165-014-0326-7

Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. 2011. The 1st Verified Software Competition: Experience Report. In *Proceedings of the 17th International Conference on Formal Methods* (Limerick, Ireland) *(FM'11)*.

Satoshi Kobayashi. 1997. Monad as modality. *Theoretical Computer Science* 175, 1 (1997). https://doi.org/10.1016/S0304-3975(96)00169-7

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (2018). https://doi.org/10.1145/3236772

K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inform. Process. Lett.* 93, 6 (2005). https://doi.org/10.1016/j.ipl.2004.10.015

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

K. R. M. Leino, P. Müller, and J. Smans. 2009. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V (Lecture Notes in Computer Science, Vol. 5705)*, A. Aldini, G. Barthe, and R. Gorrieri (Eds.). Springer-Verlag, 195–222.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (July 2019), 29 pages. https://doi.org/10.1145/3341708

Gregory Malecha. 2015. *Extensible Proof Engineering in Intensional Type Theory*. Ph.D. Dissertation. https://dash.harvard.edu/handle/1/17467172

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008). https://doi.org/10.1017/S0956796807006326

Carroll Morgan. 1988. The Specification Statement. *ACM Trans. Program. Lang. Syst.* 10, 3 (1988). https://doi.org/10.1145/44501.44503

Carroll Morgan. 1990. *Programming from specifications*. Prentice-Hall, Inc.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning *(Lecture Notes in Computer Science)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, Berlin, Heidelberg, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Greg Nelson. 1989. A Generalization of Dijkstra's Calculus. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 517–561. https://doi.org/10.1145/69558.69559

Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with applications: an appetizer*. Springer Science & Business Media.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2009. Separation and Information Hiding. *ACM Trans. Program. Lang. Syst.* 31, 3, Article 11 (2009). https://doi.org/10.1145/1498926.1498929

Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham.

John C. Reynolds. 2000. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in honour of Sir Tony Hoare*, J. Davies, B. Roscoe, and

J. Woodcock (Eds.). Macmillan Education UK.

Alex K Simpson. 1994. The proof theory and semantics of intuitionistic modal logic. (1994).

Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (2019). https://doi.org/10.1145/3341690

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2491956.2491978

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 89 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133913

W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967). https://doi.org/10.2307/2271658

Thomas Tuerk. 2009. A Formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg.

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code. *Proc. ACM Program. Lang.* 3, ICFP, Article 84 (July 2019), 29 pages. https://doi.org/10.1145/3341688

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, Article 58. https://doi.org/10.1145/2393596.2393665

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2009. A Machine Checked Soundness Proof for an Intermediate Verification Language. In *SOFSEM 2009: Theory and Practice of Computer Science*, Mogens Nielsen, Antonín Kučera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tůma, and Frank Valencia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2010. A Machine-Checked Soundness Proof for an Efficient Verification Condition Generator. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1774088.1774610

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.