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

Anonymous Author(s)

Abstract

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 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 relate the resulting symbolic executor with the concrete verification condition generator using a Kripke logical relation. We report on the practical application of these techniques in JibBoom, 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 concrete 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 JibBoom, we can report good results on practicality and efficiency of the tool, demonstrating practical viability of our symbolic execution approach.

Keywords: program verification, symbolic execution, predicate transformers, separation logic, refinement, logical relations

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 [25] 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 or weakest preconditions to decide largely automatically whether a program satisfies the program logic rules. For many tools like Dafny[38], VeriFast [31] etc. this 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 Verismall [4], VST-Floyd [14], etc. 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 manipulate assertions representing intermediate states of different execution paths of the program. For efficiency reasons, it is important to simplify these assertions during verification, and eagerly prune unreachable states. At the same time, proving soundness requires justifying such simplifications and proving that the simplified assertions still cover all possible concrete execution states.

To address this challenge, many tools shallowly embed intermediate assertions as meta-logic properties and make use of the meta-logic's meta-programming facilities to conveniently implement assertion simplification and pruning. 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 tools have managed this: VeriSmall [4] and Featherweight VeriFast [32].However, the former features a

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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 sound VC generators by writing interpreters in concrete and symbolic predicate transformer monads.
- We define a Kripke frame to make contextual information (path constraints) 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, as a postprocessing of the tree representation of the verification condition resulting from symbolic execution.
- We show how symbolic VC generation can be proved sound w.r.t. concrete VC generation by means of a novel logical relation.
- We demonstrate the reusability of our approach by implementing VC generators for Jib[1]. The latter forms the basis for JibBoom, 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 concrete 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 concrete ones, by constructing a Kripke-indexed logical relation between the two specification monads. Finally, Section 6 explains JibBoom, 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 (VC) 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 post condition:

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

[1]Anonymized for review.

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

Figure 1. Object language syntax

The weakest precondition operator maps a statement s to a predicate transformer, which in turns 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 [2, 30, 50].

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 answer type $\mathbb{P}$

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

also called the *backwards predicate transformer monad* [40].

This has been exploited in the F* language [49] to index effectful monadic computations with their semantics as predicate transformers, allowing the user to co-design programs and specifications. Furthermore, [40] 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 [39] 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 Jib, the language that JibBoom works with (see Sec. 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 represented directly as propositions in the host language, i.e. a shallow embedding. The next section focuses on deep embeddings, i.e. symbolic representations.

Figure 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 [34]: 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:

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

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.

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

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

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

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

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

Figure 3. Weakest precondition for control-flow branches

and guards for local assumptions and assertions [20, 32, 46]. Figure 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 Figure 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, Figure 3, where we show the interpretation of if expressions and pattern matching for sum types. The definition uses a function $matchbool_\otimes$ which uses $\otimes$ and assume to implement pattern matching on booleans. Pattern matching on sum types is implemented similarly, except that

$$exec\ (x := e) := eval\ e >\!\!>\!\!= assign\ x$$
$$eval\ e : M\ v := \lambda\delta\ post.\ post\ (eval\_exp\ e\ \delta)$$
$$assign\ x\ v : M\ () := \lambda\delta\ post.\ post\ (update\ \delta\ x\ v)\ ()$$

Figure 4. Weakest precondition for mutable variable assignments

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 post condition Q is normalized first before continuing the recursion over the statement s. However, due to the higher-order representation of predicates in W_δ, recursion over e is happening first and the post condition 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 [27–29, 57].

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 [1] 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 [3]. 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 $x := x + x; x := x + x; x := x + x$ 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:

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

2.3 Functions

To verify function, we need a way to declare their specifications. For this, we define a form of contracts that specify pre and post conditions. In the anticipation of moving to deep embeddings in the next section, we already define parts

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

Figure 5. Deeply-embedded values

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

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

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 prevents conflating logic and program variables, which would be necessary if we specified contracts with program variables + ghost variables. It also gives us a bit more flexibility, because patterns can be non-linear. Often the pattern for a program variable will just be a logic variable. RES specifies a logic variable ℓ for the result value. The pre condition REQ is a predicate on Σ-valuations and the post condition a predicate on (Σ, ℓ)-valuations. In the following, we will skip the repetition of the formal parameters $\overline{x}$ and also refer to 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 the product of the computed maximum and length.

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

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

Figure 6. Weakest precondition for function calls

$$\frac{\{P\}\ e; \delta\ \{R\} \quad \forall \delta'.\ \{R\ \text{true}\ \delta'\}\ e_1; \delta'\ \{Q\} \quad \forall \delta'.\ \{R\ \text{false}\ \delta'\}\ e_2; \delta'\ \{Q\}}{\{P\}\ \text{if } e \text{ then } e_1 \text{ else } e_2; \delta\ \{Q\}}$$

$$\frac{P \vdash P' \qquad \forall v, \delta'.\ Q'\ v\ \delta' \vdash Q\ v\ \delta' \qquad \{P'\}\ s; \delta\ \{Q'\}}{\{P\}\ s; \delta\ \{Q\}}$$

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

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

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

Function calls are verified by interpreting the contract as a specification statement [43], as defined in Fig. 6.

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

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

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

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

When we move to a symbolic executor in Section 3, this definition will allow us to seed the wp calculation with the precondition, a technique also known as *preconditioned symbolic execution* and to let the executor try to solve proof 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. 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 [14] and [41]. An excerpt of the axioms is depicted in Figure 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 [33] model against the operational semantics of Jib.

In terms of this program logic, we prove soundness of our concrete verification conditions in the following lemma:

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

$$\{\text{True}\}\ e; \delta\ \{post\}.$$

Alternatively phrased, the following triple always holds:

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

This 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., 47]. Specifically, FVF [32] 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 verification condition generator from Section 2 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

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

Figure 8. Deeply-embedded formulas and propositions

interpreter based on symbolic representations (deep embeddings) that achieves this. Note that the result of running this interpreter is still a proposition, albeit with 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. Is an asserted formula inconsistent with the path condition, we emit a `False` proposition.

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 $(A \to \mathrm{env} \to \mathbb{S}) \to (\mathrm{env} \to \mathbb{S})$ that generates propositions in a symbolic universe of propositions $\mathbb{S}$ rather than meta-language propositions in Prop. Additionally, the use of symbolic propositions requires careful bookkeeping of the logical variables in scope and the path condition. To deal with this bookkeeping in a principled manner, we define a Kripke frame to classify values and computations with the logical variable scope and path constraints in which they are meaningful, and to enforce monotonicity of computations. Finally, our symbolic execution interacts with a constraint solver during execution, 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 Sections 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 the interaction with the constraint solver in Section 3.6 and 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 cannot programmatically inspect values or propositions during execution, for instance for semi-automatic simplification or solving. Instead we develop a deep embedding. Fig. 8 contains definitions.

$$\mathrm{solver} : \mathbb{C} \to \overline{\mathbb{F}} \to \mathrm{option}(\zeta, \overline{\mathbb{F}})$$
$$\mathrm{solver}\ \mathbb{C}\ \overline{\mathbb{F}_0} = \mathrm{Some}(\overline{l \mapsto t}, \overline{\mathbb{F}_1}) \quad \Leftrightarrow (\mathbb{C} \vdash \overline{\mathbb{F}_0} \Leftrightarrow \mathbb{C} \vdash \overline{l = t} \wedge \overline{\mathbb{F}_1})$$
$$\mathrm{solver}\ \mathbb{C}\ \overline{\mathbb{F}_0} = \mathrm{None} \quad \Leftrightarrow \mathbb{C} \vdash \neg\overline{\mathbb{F}_0}$$

Figure 9. Solver interface

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 $\exists l.\mathbb{S}$ and $\forall l.\mathbb{S}$ for arbitrary choice. We discuss the debug production in Section 3.5 and the last two productions at the end of the next section.

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 $\overline{F'}$, s.t. $\mathbb{C} \vdash \mathbb{F}_0 \Leftrightarrow \mathbb{C} \vdash \overline{F'}$. We could therefore replace the original problem F_0 with the sub-problem $\overline{F'}$ 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 logical 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 [6]) $\zeta = \overline{l \mapsto t}$ 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 $(l \mapsto t) \to \mathbb{S}$ and assert $(l \mapsto t) \wedge \mathbb{S}$. In both cases, the executor can eliminate the variable from further consideration by

$$
\begin{array}{ll}
w & := \{(\Sigma, \mathbb{C}) \mid \mathrm{fv}(\mathbb{C}) \subseteq \Sigma\} \\
(\Sigma_1, \mathbb{C}_1) \sqsupseteq (\Sigma_2, \mathbb{C}_2) & := \{\zeta \in \mathrm{Sub}[\Sigma_1, \Sigma_2] \mid \mathbb{C}_2 \vdash \mathbb{C}_1[\zeta]\} \\
\hline
\vdash A & := \forall w, A\ w \\
A \to B & := \lambda w. A\ w \to B\ w \\
\Box A & := \lambda w. \forall w', w \sqsupseteq w' \to A\ w' \\
\hline
\mathbf{K} & : \vdash \Box(A \to B) \to \Box A \to \Box B := \\
\multicolumn{2}{l}{\quad \lambda w\ f\ a\ w'\ (\omega : w \sqsupseteq w').f\ w'\ \omega\ (a\ w'\ \omega)} \\
\mathbf{T} & : \vdash \Box A \to A := \lambda\ w\ a.a\ w\ \mathrm{id}_{\sqsupseteq} \\
\mathbf{4} & : \vdash \Box A \to \Box\Box A := \\
\multicolumn{2}{l}{\quad \lambda w_0\ a\ w_1\ (\omega_1 : w_0 \sqsupseteq w_1)\ w_2\ (\omega_2 : w_1 \sqsupseteq w_2).} \\
\multicolumn{2}{l}{\qquad a\ w_2\ (\omega_2 \circ_{\sqsupseteq} \omega_1)} \\
\langle\cdot\rangle_A & : \vdash A \to \Box A \qquad A \text{ persistent}
\end{array}
$$

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

applying the substitution to all other data, e.g. the program variable store. This marks the end-of-scope [26] of the logic variable l in both productions, i.e. we require $l\#\mathbb{S}$ in both cases.

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 logical 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 Kripke frame $(w, \sqsupseteq)$ with a set w of worlds, and an accessibility relation $\sqsupseteq$ between worlds. A world is a pair of a logic variable context Σ and a path constraint that is well-formed under Σ. A world $(\Sigma_2, \mathbb{C}_2)$ is accessible from a *base world* $(\Sigma_1, \mathbb{C}_1)$ if there is a simultaneous substitution ζ for all logical variables in Σ_1 with symbolic terms in Σ_2, and under this substitution $\mathbb{C}_2$ represents a stronger set of constraints. The intuition behind this Kripke frame can be described visually: in a symbolic execution tree, each node denotes a specific world, and accessibility (over)approximates the ancestor-descendant relationship.

The universe of discourse for our deep-embedding are modal types defined as world-indexed families $w \to \mathrm{Type}$. 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.

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

$$
\begin{array}{ll}
S_{\mathrm{pure}}\ A & := \Box(A \to \mathbb{S}) \to \mathbb{S} \\
S_{\mathrm{env}}\ A & := \Box(A \to \mathrm{env} \to \mathbb{S}) \to (\mathrm{env} \to \mathbb{S}) \\
\hline
\eta & : A \to S_{\mathrm{pure}}\ A := \lambda a\ (k : \Box(A \to \mathbb{S})).\mathbf{T}\ k\ a \\
\gg\!= & : S_{\mathrm{pure}}\ A \to \Box(A \to S_{\mathrm{pure}}\ B) \to S_{\mathrm{pure}}\ B := \\
\multicolumn{2}{l}{\quad \lambda m\ (f : \Box(A \to S_{\mathrm{pure}}\ B))\ (k : \Box(B \to \mathbb{S})).m\ (\lambda a.f\ a\ (\mathbf{4}\ k))} \\
\mathrm{demonic} & : S_{\mathrm{pure}}\ t := \lambda w\ (k : \Box(t \to \mathbb{S})).\mathbf{let}\ l := \mathrm{fresh}\ w\ \mathbf{in}\ \forall l.k\ l \\
\mathrm{assume} & : \overline{\mathbb{F}} \to S_{\mathrm{pure}}\ () := \lambda w\ \overline{\mathbb{F}}\ (k : \Box(() \to \mathbb{S})). \\
\multicolumn{2}{l}{\qquad\qquad \mathbf{match}\ \mathrm{solver}\ w\ \overline{\mathbb{F}}\ \mathbf{with}} \\
\multicolumn{2}{l}{\qquad\qquad \mid \mathrm{Some}\ (\overline{l \mapsto t}, \overline{\mathbb{F}'}) \Rightarrow \overline{l \mapsto t} \to \overline{\mathbb{F}'} \to \mathbf{4}\ k\ () } \\
\multicolumn{2}{l}{\qquad\qquad \mid \mathrm{None} \Rightarrow \top}
\end{array}
$$

Figure 11. Symbolic specification monad

Viewing modal types as formulas gives rise to a Kripke model of the $\mathbf{S_4}$ modal logic, with satisfiability defined as function application $w \models A :\Leftrightarrow A\ w$. In the remainder, we will work directly in this model and use modal logic notation to structure our exposition, but otherwise not explore the logical interpretation further.

Fig. 10 (middle) contains some basic constructions. Validity $\vdash A$ means that a computation of the given type A can be used in any world. We define a type former for functions as point-wise functions of families and a box operator $\Box$, that denotes that a computation can be used in any world accessible from a base world, i.e. at a node in the execution tree and all its descendants. Fig. 10 (bottom) shows the implementation of the $\mathbf{S_4}$ axioms in our model. The **K** combinator witnesses the applicative structure [42] of the box operator $\Box$, which allows a boxed function to be applied to a boxed value in the base world instead of an accessible world. A boxed value can be immediately used in the base world via the **T** combinator, which uses the reflexivity of the accessibility relation. Accessibility is proof-relevant and the relevant part is the contained substitution. For reflexivity this is the identity substitution. The **4** combinator says that a boxed value can stay boxed when changing the base. It relies on the transitivity of accessibility which is implemented by composition of substitutions.

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 $\langle\cdot\rangle_A : \vdash A \to \Box A$. 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.

3.4 Symbolic Specification Monads

We define a backward predicate monad transformer for modal types and use it subsequently for the implementation of an interpreter. The definition of S_{pure} is in Fig. 11. Different execution branches, i.e. different worlds, will use the post condition, hence we wrap it in a $\Box$. Fig. 11 also defines some basic combinators for S_{pure}. For the purpose of presentation we

omit the explicit handling of worlds and accessibility when convenient. η implements a return and $\gg\!=$ implements a bind operator for S_{pure}. Like for the post condition in the definition of the monad, the continuation in the bind is wrapped in a $\Box$[2]. For demonic non-deterministic choice of a term, we use a fresh $: w \to l$ function that picks a locally fresh name and applies the continuation k to l in the world extended with l. In the implementation of the assume guard we call the solver on the given formulas $\overline{F}$. In the successful case, the simplification (unifications and residual formulas $\overline{F'}$) is used to construct the symbolic proposition. In the failure case, the original formulas $\overline{F}$ are inconsistent with the path condition and 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_{env} 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_{env}.

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

$$\begin{aligned}&\text{matchsum}\otimes : t \to \Box(t \to S_{\text{pure}}\ A) \to \Box(t \to S_{\text{pure}}\ A) :=\\ &\lambda t\ f\ g.\quad (x \leftarrow \text{demonic}; \text{assume}\ (\langle t\rangle = \text{inl}\ x); \mathbf{4}\ f\ x)\\ &\qquad \otimes\ (y \leftarrow \text{demonic}; \text{assume}\ (\langle t\rangle = \text{inr}\ y); \mathbf{4}\ g\ y)\end{aligned}$$

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 call pattern and path constraints that lead to the executed branch and the program variable store at that point. The path constraints are always available and the store is in the state effect that we have used so far. The initial call patterns are defined in the function contracts and live in the initial world of particular verifications, i.e. the worlds that only contain the logic variables of a contract, but no path constraints. Putting this call pattern inside an additional reader effect ensures that it is always available and updated with the substitutions of a particular branch.

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.

3.6 Post-processing

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 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 verification conditions manually. To that end we implement a post-processing 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 formula 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

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

which can be simplified to $(\forall \text{l s m}.\ldots)$, 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

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

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 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

[2]The notion we refer to is that of an $\mathcal{L}$-strong monad [35]

$$C ::= \mathcal{H}\ \overline{v} \qquad H ::= \overline{C} \qquad C ::= \mathcal{H}\ \overline{t} \qquad H ::= \overline{C}$$

Figure 12. Shallow and deep heaps and chunks

arithmetic, yields the following verification condition

$$\begin{array}{l}(\forall\ \mathrm{xs\ y\ ys.\ (xs} \mapsto \mathrm{y :: ys)} \rightarrow \\ \quad \forall\ \mathrm{l\ s\ m.s} \le \mathrm{m} * \mathrm{l} \rightarrow 0 \le \mathrm{l} \rightarrow \\ \qquad (\mathrm{m} < \mathrm{y} \rightarrow \mathrm{s} + \mathrm{y} \le \mathrm{y} * (\mathrm{l}+1) \wedge 0 \le \mathrm{l}+1 \wedge \top) \wedge \\ \qquad (\mathrm{m} \ge \mathrm{y} \rightarrow \mathrm{s} + \mathrm{y} \le \mathrm{m} * (\mathrm{l}+1) \wedge 0 \le \mathrm{l}+1 \wedge \top))\end{array}$$

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 formula related to that branch has been fully removed. For easy navigation, we decided to never simplify the universal quantification over the logic variables (xs) from the contract, so that the call pattern is visible in the VC as the unification ($\mathrm{xs} \mapsto \mathrm{y :: ys}$). The second line contains the result of assuming the post-condition 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 post-processing 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.

4 Symbolic heap separation logic

Smallfoot implements a shape analysis for data structures based on separation logic. It works on a decidable fragment[11] 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 is generally used for spatial weakest precondition rules, is absent. Furthermore, Berdine et al. [12] present a symbolic execution method for automatically proving Hoare triples in this fragment.

Many verifiers [21, 31, 45] 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 our concrete and symbolic specification monads with an additional piece of state: a concrete or symbolic separation logic heap.

$$W_{heap}\ x := (x \rightarrow \delta \rightarrow H \rightarrow \mathbb{P}) \rightarrow \delta \rightarrow H \rightarrow \mathbb{P}$$
$$S_{heap}\ x := \square(x \rightarrow \mathrm{env} \rightarrow H \rightarrow \mathbb{S}) \rightarrow \mathrm{env} \rightarrow H \rightarrow \mathbb{S}$$

These heaps have the syntax depicted in Figure 12: a list of assertion constructors applied to concrete or symbolic arguments. Note that we allow these assertion 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.

```
produce_chunk : C → SMut () :=
    λc. h ← get_heap;; put_heap(c :: h)
consume_chunk : C → SMut () :=
    λ H ts. (c1..cn) ← get_heap;
        i ← angelic;
        let (H' ts) = ci in
        if H = H' then assert (ts = ts') else fail;
        put_heap (c1 ... ci−1, ci+1 ... cn)
```

Figure 13. Producing and consuming chunks

```
append(p : ptr, q : llist) : unit :=
  lemma open_cons p;
  let mbn := foreign snd p in
  case mbn of inl n ⇒ call append n q
              inr y ⇒ lemma close_nil y;
                      foreign setsnd p q;
  lemma close_cons p
```

Figure 14. Appending two linked lists

In terms of this additional state, we can then define produce_chunk and consume_chunk functions, the symbolic versions of which are shown in Figure 13. Based on these two functions, we can define more general produce and consume functions which produce and consume arbitrary structured assertions, but we do not go into this further for space reasons. The definitions are generally similar to those used in Feitherweight VeriFast [32] and μVeriFast [18].

4.1 Example: Linked lists

To give you an idea of practical use of our separation logic solver, Figure 14 shows an append function for lists, implemented in our object language. It is implemented in terms of primitive functions mkcons, snd and setsnd, whose contracts are shown in Figure 15, together with the contract for append itself. Append's implementation is annotated with three lemma invocations, which do not have any runtime behavior, but transform the proof state according to their contracts depicted in Figure 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. In the future we are planning to support user-provided heuristics which automatically invoke such lemmas where needed.

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

$\forall p$: ptr, q : llist, xs ys : list int.	$\{\text{inl } p \mapsto_l xs * q \mapsto_l ys\}$	$append(p, q)$	$\{\_.\ \exists zs : \text{list int}.\ \text{inl } p \mapsto_l zs * \text{append}(xs, ys, zs)\}$
$\forall x$: int, xs : llist.	$\{\text{true}\}$	$mkcons(x, xs)$	$\{p.\ p \mapsto_p (x, xs)\}$
$\forall p\ x$: int, xs : llist.	$\{p \mapsto_p (x, xs)\}$	$snd(p)$	$\{r.\ r = xs * p \mapsto_p (x, xs)\}$
$\forall p\ x$: int, xs : llist.	$\{\exists ys : \text{llist}.\ p \mapsto_p (x, ys)\}$	$setsnd(p)$	$\{\_.\ p \mapsto_p (x, xs)\}$

Figure 15. Contracts for linked lists functions

$\forall p$: int, xs : list int.	$\{\text{inl } p \mapsto_l xs\}$	$open\_cons(p)$	$\left\{\begin{array}{l}\text{match } xs \text{ with}\\ \quad \mid \text{nil} \mapsto \bot\\ \mid \text{cons}(y, ys) \mapsto \exists n : \text{llist}.\ p \mapsto_p (y, n) * n \mapsto_l ys\end{array}\right\}$
$\forall \begin{array}{l}p : \text{ptr}, x : \text{int},\\ xs : \text{list int}, n : \text{llist}\end{array}$	$\{p \mapsto_l (x, n) * n \mapsto_l xs\}$	$close\_cons(p)$	$\{\text{inl } p \mapsto_l \text{cons}(x, xs)\}$
$\forall p$: unit, xs : list int.	$\{p \mapsto_p (x, n) * n \mapsto_l xs\}$	$close\_nil$	$\{p = \text{unit} * xs = \text{nil}\}$

Figure 16. Lemmas about linked list heap predicates

outputs is dictated by our soundness requirement: given a program, the (closed) symbolic VC should imply the concrete one $(\epsilon \models \mathbb{S}) \to \mathbb{P}$.

We have to generalize this in two ways. First, we need to consider other moments of the execution, instead of just the final result. That means we are in a world w and want to consider the implication $(\iota_w \models \mathbb{S}) \to \mathbb{P}$ 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 [52] $\mathcal{R}_\lesssim$ 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 [8, 44].

The refinement relation is defined in Fig. 17. $\mathcal{R}_\lesssim[\![AT, A]\!]$ relates symbolic computations of type AT 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 $\omega : w \sqsupseteq w'$, 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 accessbility.

Soundness. The soundness of the symbolic VC gen can now be stated as the inclusion in the refinement relation in the empty (initial) world:

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

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

$$(w, \iota_w, \text{EXEC } s, \text{exec } s) \in \mathcal{R}_\lesssim[\![S_{\text{env}}\ t, W_{\text{env}}\ v]\!]$$
$$(w, \iota_w, \gg\!=, \gg\!=) \in \mathcal{R}_\lesssim[\![S\ A \to \Box(A \to S\ B) \to S\ B, \ldots]\!]$$

This is mostly straightforward, 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 consider the demonic choice combinators (shown in Figures 2 and 11). After inlining the definitions, applying them to two related post conditions

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

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

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

After using relatedness of the post conditions it remains to show that the logic variable ℓ is related to v in $(\iota_w, \ell \mapsto v)$ which is immediate.

6 JibBoom

JibBoom is a verifier for Sail [5] that implements the techniques described in this paper. Sail is a domain-specific language for executable specifications of instruction set architectures and JibBoom implements a variant of Sail called Jib, which is deeply embedded in the Coq theorem prover. Sail and Jib feature structured types such as lists, enums, records unions and bit-vectors[3]. The goal of JibBoom is to support semi-automatic proofs of ISA security properties, something we will report in more detail elsewhere.

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

$$\mathcal{R}_{\lesssim}[\![A, a]\!] \subseteq \{(w, \iota_w, A, a)\}$$
$$\mathcal{R}_{\lesssim}[\![\mathrm{V}, \mathrm{v}]\!] = \{(w, \iota_w, V, v) \mid v = V[\iota_w]\}$$
$$\mathcal{R}_{\lesssim}[\![\mathbb{S}, \mathbb{P}]\!] = \{(w, \iota, \mathbb{S}, \mathbb{P}) \mid (\iota \models \mathbb{S}) \Rightarrow \mathbb{P}\}$$
$$\mathcal{R}_{\lesssim}[\![\Box A, a]\!] = \{(w, \iota, A, a) \mid \forall w', \omega : w \sqsupseteq w', \iota'.\ \text{if } \iota = [\omega.\mathrm{sub}]_{\iota'} \text{ and } [\omega.\mathrm{pc}]_{\iota'} \text{ then } (w', \iota', t, v) \in \mathcal{R}_{\lesssim}[\![A, a]\!]\}$$
$$\mathcal{R}_{\lesssim}[\![A \to B, a \to b]\!] = \{(w, \iota, f_s, f_c) \mid \forall (w, \iota, v_s, v_c) \in \mathcal{R}_{\lesssim}[\![A, a]\!].\ (w, \iota, f_s\ v_s, f_c\ v_c) \in \mathcal{R}_{\lesssim}[\![B, b]\!]\}$$

Figure 17. Refinement relation to prove soundness of symbolic execution – selected rules

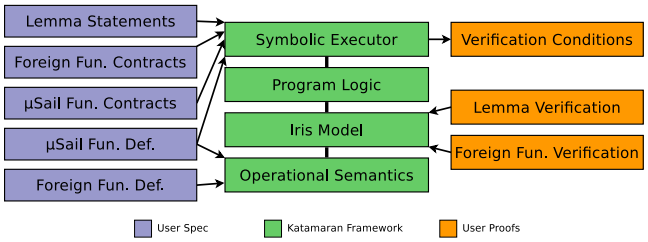

Figure 18. Structure of JibBoom

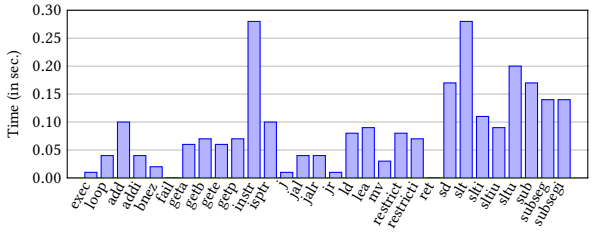

Figure 19. Verification timings for non-trivial functions in the MinimalCaps case study (run on an AMD Ryzen 7 4700U processor with 16GB DDR4-3200 SDRAM).

The structure of JibBoom is depicted in Fig. 18. The semantics of Jib is defined both in terms of a small-step operational semantics and an axiomatic program logic interface based on separation logic Hoare triples and 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 [33] 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 Jib but implemented in Coq. On top, we have a symbolic VC gen that is proved sound w.r.t. a concrete one, which in turn is proved sound to the program logic. For the symbolic executor, JibBoom 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 programming techniques [].

Jib 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 [36] for the included model.

The implementation uses an intrinsically-typed representation [7**?** **?**], 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 [22] report on similar benefits of integrating a typechecker in a verifier.

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.

Evaluation. We have instantiated our approach in our capability machine case study, called MinimalCaps [17], for which we prove that the capability safety property [19, 24, 51, 54] holds. We limit the discussion of our case study to the evaluation of JibBoom in verifying that the contracts hold. We present the results of timing contract verification for the non-trivial functions of our case study in Figure 19.

As the figure shows, contract verification using JibBoom's symbolic executor succeeds in less than 300ms for all non-trivial functions. These verification times allow us to experiment with definitions in our case study and immediately experiment with the verification part as well, enabling us to continously expand our case study and verify the corresponding contracts with JibBoom. The longer verification times in Figure 19 are explained due to the additional branches that need to be explored in the corresponding functions. The amount of branches that will be considered are kept small due to the path pruning performed by JibBoom.

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* [9, 10] or *proof by computational reflection* [13].

Existing work has already mechanized various subsets of the pipeline of verifiers. For instance, [55] contains a mechanized formalization of an intermediate verification languages (IVL), together with a proof of soundness of its VCgen, and [56] goes even further to mechanize an algorithm for compact VCs [23, 37]. 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, [4] 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 [48] 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 [14, 16, 36, 53] means. This approach has been used support large subsets of realistic programming languages such as C [14] or pure Caml [15].

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 [4] 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 answer type. Featherweight VeriFast (FVF) [32] uses an intensional specification monad that supports angelic and demonic choice, which is not based on continuation monads.

Dijkstra monads [2, 30, 40, 50] 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 Sec. 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 [4] 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 [32] 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: post condition 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

[1] 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

[2] 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 (POPL 2017)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3009837.3009878

[3] 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.

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

[5] 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

[6] 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

[7] 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

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

[9] 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

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

[11] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. 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.

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

[13] 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.

[14] 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

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

[16] 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

[17] MinimalCaps Developers. 2021. *MinimalCaps Case Study*. https://github.com/katamaran-project/minimalcaps

[18] 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

[19] 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

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

[21] 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

[22] 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.

[23] 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 (POPL '01)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/360204.360220

[24] 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

[25] 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

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

[27] 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.

[28] 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

[29] 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

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

[31] 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.

[32] 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

[33] 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

[34] 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 (FM'11)*.

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

[36] 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

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

[38] 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.

[39] 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 (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[40] 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

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

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

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

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

[45] 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

[46] 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

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

[48] 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.

[49] 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.

[50] 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 (PLDI '13)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2491956.2491978

[51] 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

[52] 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

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

[54] 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

[55] 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.

[56] 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 (SAC '10)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1774088.1774610

[57] 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.

A Sum-max-len example in the shallow executor

```
forall xs : list Z, true = true ->
(nil = xs -> exists (v : Z * Z) (v0 : Z), (v, v0) = ((0, 0), 0) /\
 (exists v1 v2 : Z, (v1, v2) = v /\ v1 <= v2 * v0 /\ 0 <= v0 /\ True)) /\
(forall (v : Z) (v0 : list Z), (v :: v0) = xs -> exists v1 : list Z, [v1] = [v0] /\ true = true /\
 (forall (v2 : Z * Z * Z) (v3 : Z * Z) (v4 : Z), (v3, v4) = v2 -> forall v5 v6 : Z, (v5, v6) = v3 ->
  v5 <= v6 * v4 -> 0 <= v4 -> forall (v7 : Z * Z) (v8 : Z), (v7, v8) = v2 ->
  forall v9 v10 : Z, (v9, v10) = v7 ->
   ((v10 <? v) = true -> exists (v11 : Z * Z) (v12 : Z), (v11, v12) = ((v9 + v, v), v8 + 1) /\
      (exists v13 v14 : Z, (v13, v14) = v11 /\ v13 <= v14 * v12 /\ 0 <= v12 /\ True)) /\
   ((v10 <? v) = false -> exists (v11 : Z * Z) (v12 : Z), (v11, v12) = ((v9 + v, v10), v8 + 1) /\
      (exists v13 v14 : Z, (v13, v14) = v11 /\ v13 <= v14 * v12 /\ 0 <= v12 /\ True))))
```

Figure 20. The `sum-max-len` example with the shallow executor running the symbolic contracts.

```
forall xs : list Z, true = true ->
(nil = xs -> (0 <= 0 * 0 /\ 0 <= 0) /\ True) /\
(forall (v : Z) (v0 : list Z), (v :: v0) = xs ->
 exists v1 : list Z, [v1] = [v0] /\ true = true /\
   (forall v2 : Z * Z * Z,
    (let '((s, m), l) := v2 in s <= m * l /\ 0 <= l) ->
    forall (v3 : Z * Z) (v4 : Z), (v3, v4) = v2 ->
    forall v5 v6 : Z, (v5, v6) = v3 ->
    ((v6 <? v) = true -> (v5 + v <= v * (v4 + 1) /\ 0 <= v4 + 1) /\ True) /\
    ((v6 <? v) = false -> (v5 + v <= v6 * (v4 + 1) /\ 0 <= v4 + 1) /\ True)))
```

Figure 21. The `sum-max-len` example with the shallow executor running on shallow contracts with pattern matching.

```
forall v : list Z, true = true ->
(nil = v -> (forall s m l : Z, ((0, 0), 0) = ((s, m), l) -> s <= m * l /\ 0 <= l) /\ True) /\
(forall (v0 : Z) (v1 : list Z), (v0 :: v1) = v ->
 exists v2 : list Z, [v2] = [v1] /\ true = true /\
   (forall v3 : Z * Z * Z,
    (forall s m l : Z, v3 = ((s, m), l) -> s <= m * l /\ 0 <= l) ->
    forall (v4 : Z * Z) (v5 : Z), (v4, v5) = v3 ->
    forall v6 v7 : Z, (v6, v7) = v4 ->
    ((v7 <? v0) = true -> (forall s m l : Z, ((v6 + v0, v0), v5 + 1) = ((s, m), l) -> s <= m * l /\ 0 <= l) /\ True) /\
    ((v7 <? v0) = false -> (forall s m l : Z, ((v6 + v0, v7), v5 + 1) = ((s, m), l) -> s <= m * l /\ 0 <= l) /\ True)))
```

Figure 22. The `sum-max-len` example with the shallow executor running on shallow contracts without pattern matching.