FO-Complete Program Verification for Frame Logic

ADITHYA MURALI, University of Illinois Urbana-Champaign, USA
HRISHIKESH BALAKRISHNAN, University of Illinois Urbana-Champaign, Department of Computer Science, USA
AARON COUNCILMAN, University of Illinois Urbana-Champaign, Department of Computer Science, USA
P. MADHUSUDAN, University of Illinois Urbana-Champaign, Department of Computer Science, USA

We develop techniques for automating program verification for specifications in frame logic. Frame Logic is a logic for specifying properties of heap manipulating programs that is based on first order logic with recursive definitions (FORD) that has a support operator akin to implicit heaplets in separation logic. We develop a FO-complete automatic verification technique for programs annotated with frame logic specifications using a novel verification condition generation followed by reasoning in FORD using the technique of natural proofs. We implement a tool that realizes our technique and show its efficacy on a suite of benchmarks that manipulate data structures. We also develop a separation logic with an alternate semantics that can be converted to frame logic to realize FO-complete program reasoning.

1 INTRODUCTION

Automated verification of programs that destructively manipulate heaps remains a challenging open problem. Separation logic has emerged as a popular specification logic for expressing properties of structures in heaps. While separation logic is used extensively in interactive theorem proving settings, automation of separation logic reasoning has not yet met the same level of success.

One problem for automation of separation logic is that several aspects of it are inherently *second-order*, making it extremely hard to automatically reason with, even incompletely, using engines such as SMT solvers that are based on first-order logic. In particular, the *magic wand* ($-\!\!*$) in separation logic quantifies over *arbitrary heaps* that satisfy a property, which is inherently second-order [Brochenin et al. 2008]. The magic wand arises in many settings in program verification, especially in order to express weakest preconditions and in expressing loop invariants [O'Hearn 2012; Reynolds 2002]. Separation logic reasoning tools evaluated in competitions (like SLComp [Sighireanu 2021]) seldom support the magic-wand operator and attest to the difficulty of reasoning with it. Even if the magic wand is avoided (like using incomplete strongest post calculations, like in [Berdine et al. 2005]), expressing properties such as *separating conjunction*, *conjunction*, and *disjunction* requires quantifying over sets of locations, and when proving implication or entailment, result in formulas with both existential and universal quantification over sets that are hard to reason with.

Frame Logic. Recently, a new logic called *frame logic* (FL) was proposed by Murali et al. that embraces the *principles* of separation logic but works in a *first-order logic setting with recursive definitions (FORD)* [Murali et al. 2023, 2020]. Frame logic (FL) is aesthetically very simple— it adds to first-order logic with recursive definitions a *support* operator *Sp*, where $Sp(\alpha)$, for any formula α, evaluates to a subset of locations of the heap that the truthhood/falsehood of α relies upon. The support is similar to local heaplets in separation logic. However, unlike separation logic, supports are uniquely defined, which allows modeling them without true quantification.

Instead of relying on a localized heaplet semantics as separation logic does, frame logic keeps the classical semantics of FORD over the *global heap*, but allows recovering the local heaplet using the support operator. Frame logic can express the separating conjunct — $\alpha * \beta$ is essentially expressed as $\alpha \wedge \beta \wedge Sp(\alpha) \cap Sp(\beta) = \emptyset$. It also supports frame reasoning: if α holds in a heap and $Sp(\alpha)$ is not modified by a program mutating the heap, then α continues to hold.

The primary goal of this paper is to develop automated SMT-based approaches for reasoning with programs annotated with frame logic specifications.

Work on frame logic by Murali et al [Murali et al. 2020] has argued that weakest preconditions for programs with frame logic annotations are expressible in frame logic itself, and that verification conditions in frame logic can be translated to FORD. However, this does not translate to automation as FORD with alternating quantifiers is highly complex. The weakest precondition transformations described in [Murali et al. 2023, 2020] are complex, involving several introductions of *quantifiers* (including existential quantification) leading to weakest preconditions having alternations of universal and existential quantification. Furthermore, the weakest preconditions introduce first-order formulations of magic wand ("MW-operators") that have gnarly definitions that are too complex to provide automation for.

Automation using Natural Proofs and the FO-Complete Fragment $L_{\mathbf{oneway}}$. In this paper, we approach automated verification of frame logic afresh with the goal of embedding verification conditions in a known automatable class of first-order logic with recursive definitions (FORD). The technique of natural proofs is a well-established technique for handling such logics— it uses recursive definition unfoldings, uninterpreted function abstractions, and SMT-based quantifier-free reasoning to build sound automated reasoning [Pek et al. 2014; Qiu et al. 2013]. Furthermore, the work on its foundations [Löding et al. 2018] identifies particular fragments for which the natural proofs technique is FO-complete. More precisely, the work shows that when recursive definitions are abstracted to be *fixpoints* rather than least fixpoints, the properties are expressible in first-order logic and natural proofs heuristics are complete for this interpretation. The work in [Löding et al. 2018] posits that the efficacy in practice of the natural proofs technique is perhaps due to the completeness of this technique.

Our goal is to provide automated reasoning for programs with frame logic specifications that is also FO-complete in the above sense. In order to achieve this, we need to generate verification conditions that are sound and complete, and also ensure that the verification conditions are in the fragment L_{oneway} for which natural proofs are known to be FO-complete. L_{oneway} has an *uninterpreted foreground sort* and multiple background sorts and has two restrictions: (a) existential quantification is allowed only over the foreground sort, and (b) uninterpreted functions from background sorts to the foreground sort are disallowed (functions are "one-way").

Generating Verification Conditions in $L_{\mathbf{oneway}}$. The first contribution of this paper is to build a program verification paradigm for reasoning with programs with frame logic annotations that generates verification conditions in the logic L_{oneway}. This leads to FO-complete automation using the natural proofs technique. We overcome several challenges for obtaining such a translation using novel ideas that we describe below.

First, we have to choose a frame logic that disallows quantification. However, recursive definitions of data structures typically use existential quantification in order to recurse on smaller data structures. We introduce a new `cloud` operator ($[\cdot]$) that removes a special form of existential quantification our logic provides.

Second, we need to generate verification conditions with care so that they are in the L_{oneway} fragment and have existential quantification only over the foreground sort. The verification conditions are generated using strongest-post like symbolic evaluation rather than weakest preconditions. We also introduce *frame reasoning* when heaps are updated using universally quantified FO formula in L_{oneway} (earlier work on frame logic did not do any frame reasoning [Murali et al. 2023, 2020]). Frame reasoning is crucial when using techniques such as natural proofs as they allow simpler ways to show properties are maintained when heaps are modified, which otherwise may require induction and evade natural proofs reasoning. The earlier work on frame logic [Murali et al. 2023, 2020] did not address handling of function calls, and we show new mechanisms to capture "havoc"-ing the implicit heaplet (support of the precondition of the called function) and associated frame reasoning while staying within the logic L_{oneway}. Finally, our program verification framework also handles memory safety using implicit supports— the allocated set is assumed to be the support of the precondition and we check that all dereferences are within the allocated set.

The Frame Logic Verifier (FLV) Tool and Evaluation. The second contribution of this paper is an implementation and evaluation of our automated verification technique for a simple programming language that destructively updates heaps against frame logic specifications. Our tool FLV (Frame Logic Verifier) generates verification conditions in first-order logic with recursive definitions, in particular in the logic L_{oneway}, and utilizes an existing tool to do natural proofs reasoning [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013]. We allow users to write additional lemmas that they believe can be proved using the lemma itself as the induction hypothesis, and the tool converts the required inductive proof (using the pre-fixpoint) and disposes its verification using natural proofs.

We evaluate our verification methodology by utilizing the FLV tool to verify a suite of programs that manipulate common data structures. The specifications written in frame logic are fairly complete functional specifications of the methods. Furthermore, we share the experience of one of the co-authors, who was not involved in the development of the tool, but who wrote all the programs and specifications in frame logic, especially in writing support specifications, debugging specifications, and the interaction methodology and experience with the tool.

New Pathways for Automation of Separation Logic:. We posit that our work can pave new pathways for automating other logics, like separation logic, through a translation to frame logic. As we argued earlier, standard semantics for separation logic makes it hard to automate using FO solvers. The third contribution of this paper is a separation logic with *alternate semantics inspired by frame logic* (*SL-FL*) that ensures determined heaplets. We show that this separation logic can be translated to frame logic, where the tight heaplets in separation logic correspond to supports in frame logic. Utilizing the automation for frame logic that we have developed, this shows that the new separation logic is amenable to FO-complete program verification as well.

Summary. We propose techniques and tools that provide automation in reasoning with programs against specifications written in frame logic, a powerful logic that extends first-order logic with recursive definitions with implicit supports. The primary contributions are:

- A universal fragment of frame logic that allows restricted forms of guarded existential quantification that can be removed using a novel cloud operator.
- An automated program verification methodology that uses verification condition generation in the logic L_{oneway}, which is automated using natural proofs and SMT reasoning that provides *FO*-complete reasoning.
- The development of the tool FLV that realizes the verification methodology, and an evaluation of the technique and the tool on a suite of programs that manipulate data structures.
- A separation logic with alternate semantics that can be translated to frame logic, thereby showing FO-complete automated program verification for this separation logic.

2 FRAME LOGIC AND PROGRAM VERIFICATION

In this section, we describe some preliminaries on Frame Logic and the notion of validity for Hoare Triples that we automate in this work. Crucially, we add a new *cloud* operator $[\cdot]$ to Frame Logic that facilitates expressing properties without quantification.

2.1 First-Order Logic with Recursive Definitions (FORD)

Frame Logic extends First-Order Logic with Recursive Definitions (FORD). FORD is similar to first-order logic with least fixpoints [Aho and Ullman 1979; Chandra and Harel 1980; Immerman 1982; Libkin 2004; Vardi 1982], except recursive definitions (which have least fixpoint semantics) are given names.

Formally, we have a signature $\Sigma = (\mathcal{S}, C, \mathcal{F}, \mathcal{R}, \mathcal{I})$ where $\mathcal{S}$ is a finite set of sorts, and $C, \mathcal{F}$ and $\mathcal{R}$ are sets of constant, function, and relation symbols respectively. $\mathcal{I}$ is a set of relation symbols disjoint from $\mathcal{R}$ whose interpretations are given using recursive definitions (as opposed to being interpreted by a model). Symbols have their usual types, e.g., function symbols in $\mathcal{F}$ have an associated arity $n \in \mathbb{N}$ and are of type $\tau_1 \times \tau_2 \ldots \times \tau_n \longrightarrow \tau$, where $\tau_i, \tau \in \mathcal{S}$.

We require that $\mathcal{S}$ contain a designated *foreground* sort *Loc*. We use the foreground sort to model heap locations. The remaining sorts, called *background* sorts, are used to model data values such as integers, sets, etc. We use the function symbols in $\mathcal{F}$ to model pointers and data fields of heap locations. For example, the next pointer of a linked list can be modeled using the symbol $next : Loc \longrightarrow Loc$, and similarly, the key stored at a location can be modeled using $key : Loc \longrightarrow Int$.

We do not provide the syntax of FORD here as it is essentially identical to the syntax of Frame Logic given in Figure 1, except that FORD does not contain the support operator $Sp(\cdot)$ and cloud operator $[\cdot]$.

Recursive Definitions. A recursive definition of a predicate $I \in \mathcal{I}$ is of the form

$$I(\overline{x}) :=_{lfp} \rho(\overline{x})$$

where ρ is a quantifier-free formula that only mentions recursively defined symbols in $\mathcal{I}$ positively (i.e., under an even number of negations). This ensures that least fixpoints always exist [Tarski 1955]. We formally treat recursively defined functions by modeling them as predicates, however, we will use function symbols with recursive definitions in our exposition. We denote the set of definitions for the symbols in $\mathcal{I}$ by $\mathcal{D}$. We require that $\mathcal{D}$ contains exactly one definition for each $I \in \mathcal{I}$.

Semantics. We consider first-order models where the foreground sort *Loc* is uninterpreted and the background sorts are constrained by a first-order theory. This theory is usually the combination of several theories over individual sorts such that the quantifier-free fragment of the combination is decidable [Nelson and Oppen 1979].

Given a set of definitions $\mathcal{D}$ for the symbols in $\mathcal{I}$, a model of FORD consists of a first-order model of the above kind that interprets the symbols in C, $\mathcal{F}$, and $\mathcal{R}$ (respecting the various theories), as well as an interpretation for the symbols in $\mathcal{I}$ that is determined by the first-order model as the least fixpoint of the definitions $\mathcal{D}$. Formulas are then evaluated as usual.

The L_{oneway} Fragment. L_{oneway} is a syntactic fragment of FORD introduced in prior work [Löding et al. 2018] which allows only "one-way" functions from the foreground sort to the background sorts. Formally, every function symbol in $\mathcal{F}$ of arity n whose range sort is the foreground sort *Loc* has domain Loc^n. Recursively defined symbols $\mathcal{I}$ of arity k are of type Loc^k. Finally, formulas are only allowed to quantify existentially over *Loc* (for validity). Validity checking for formulas in L_{oneway} can be automated effectively using a systematic quantifier-instantiation procedure [Löding et al. 2018] based on Natural Proofs [Pek et al. 2014; Qiu et al. 2013] that is *complete* with respect to a fixpoint abstraction of recursive definitions (rather than the true least fixpoint semantics). In Section 3 we describe a VC generation mechanism that reduces the correctness of Hoare Triples to the validity of Frame Logic formulas. In Section 4 we describe how to translate the generated FL formulas to FORD formulas in L_{oneway}, as well as the relatively complete procedure for reasoning with them.

$$\begin{array}{rcll}
\text{FL Formulas} & \varphi & \coloneqq & \bot \mid \top \mid t = t \mid R(t_1 \ldots, t_m) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \\
& & & \mid \mathit{ite}(\gamma : \varphi, \varphi) \mid \exists y \colon y = f(x).\ \varphi \\
& & & \text{where } R \in \mathcal{R} \cup \mathcal{I},\ y \in \mathit{Var}_{Loc},\ f \in \mathcal{F}_m \\
\text{Guards} & \gamma & \coloneqq & t = t \mid R(t_1 \ldots, t_m) \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \mid \neg\gamma \mid \mathit{ite}(\gamma : \gamma, \gamma) \\
& & & \text{where } R \in \mathcal{R},\ t_i, t \text{ are not of type } \mathit{Set}(\mathit{Loc}) \\
\text{Terms} & t & \coloneqq & c \mid x \mid f(t_1 \ldots, t_m) \mid \mathit{ite}(\gamma : t, t) \\
& & & \mid \mathit{Sp}(\varphi) \mid \mathit{Sp}(t') \quad \text{if } t \text{ is of type } \mathit{Set}(\mathit{Loc}) \\
& & & \text{where } c \text{ is a constant, } x \text{ is a variable of the appropriate type}
\end{array}$$

Fig. 1. Frame Logic with guarded quantification. Var_{Loc} denotes variables over the foreground sort and $\mathit{Set}(\mathit{Loc})$ denotes the sort consisting of sets of foreground elements. Terms and formulas are assumed to be well-typed for simplicity of presentation.

2.2 Frame Logic with Guarded Quantification

Frame Logic (FL) [Murali et al. 2023, 2020] extends FORD with a *support* operator $\mathit{Sp}(\cdot)$. For a formula φ (term t), $\mathit{Sp}(\varphi)$ denotes the subset of locations (foreground elements) on which the truth of φ (resp. value of t) depends. In the context of heaps, the support can be thought of as the "heaplet" of φ. FL uses the Sp operator to state disjointness and reason about framing, e.g., the formula $\mathit{List}(x) \wedge \mathit{List}(y) \wedge \mathit{Sp}(\mathit{List}(x)) \cap \mathit{Sp}(\mathit{List}(y)) = \emptyset$ says that x and y point to disjoint linked lists.

Syntax. In this work, we use a fragment of FL with guarded quantification shown in Figure 1. Formally, we distinguish a set of *mutable functions* $\mathcal{F}_m$ among the symbols in $\mathcal{F}$ with domain Loc that model pointer and data fields over the foreground sort. We also require a background sort $\mathit{Set}(\mathit{Loc})$ representing sets of foreground locations, and define $\mathit{Sp}(\varphi)$ as a term of type $\mathit{Set}(\mathit{Loc})$. We utilize in the syntax *ite* ("if-then-else") expressions over terms and formulas, where $\mathit{ite}(\gamma : \alpha, \beta)$ denotes "if γ holds then α else β". Note that the *guard* γ cannot mention inductively defined predicates or terms of type $\mathit{Set}(\mathit{Loc})$, including support expressions. These are technical restrictions that are required to define a well-defined semantics for Frame Logic formulas.

We also allow formulas with *guarded* quantification of the form $\exists y \colon y = f(x).\ \varphi(y)$ where y is a variable over the foreground sort Loc. The truth value of this formula is the same as $\exists y \colon y = f(x) \wedge \varphi(y)$, but its support is defined more carefully. We describe this below.

Semantics. Since FL merely extends FORD with the $\mathit{Sp}(\cdot)$ operator, we elaborate on the semantics of Sp here. We define this as a set of recursive equations in Figure 2. Given a model M, the support of a formula α in M (resp. term), denoted $[\![\mathit{Sp}(\alpha)]\!]_M$, is the least fixpoint of the equations in Figure 2.

The support can be understood as the set of locations (i.e., "heaplet") on which mutable functions must be applied (i.e., dereferenced) in order to compute the value of a given term or formula. The support of constants is empty. The support of a term $f(t)$ in M for a mutable function $f \in \mathcal{F}_m$ is, as expected, $\{[\![t]\!]_M\}$. The application of a non-mutable function does not contribute to the support.

The support of $\alpha \wedge \beta$ is intuitively the union of the supports of α and β, and this is indeed the case in FL. Note, however, that the support of $\alpha \vee \beta$ is also the union of the supports of α and β. This is because Frame Logic defines a *unique* support for a formula *regardless of its truth value*. We can also see this reflected in the definition of $\mathit{Sp}(\neg\alpha)$, which is equal to $\mathit{Sp}(\alpha)$. This is different from, say, Separation Logic [Demri and Deters 2015; O'Hearn 2012; Reynolds 2002], where the heaplet of $\alpha \vee \beta$ is the heaplet of any of the disjuncts that evaluate to true (formulas can have multiple heaplets). Unique heaplets are a salient feature of FL, and the logic makes several design decisions to achieve this. We point the reader to prior work [Murali et al. 2023] for a discussion on the ramifications of these design decisions.

$$\llbracket Sp(c)\rrbracket_M = \llbracket Sp(x)\rrbracket_M = \emptyset \text{ for constant } c\text{, variable } x$$
$$\llbracket Sp(\top)\rrbracket_M = \llbracket Sp(\bot)\rrbracket_M = \emptyset$$
$$\llbracket Sp(Sp(t))\rrbracket_M = \llbracket Sp(t)\rrbracket_M$$
$$\llbracket Sp(t_1 = t_2)\rrbracket_M = \llbracket Sp(t_1)\rrbracket_M \cup \llbracket Sp(t_2)\rrbracket_M$$
$$\llbracket Sp(\alpha \wedge \beta)\rrbracket_M = \llbracket Sp(\alpha)\rrbracket_M \cup \llbracket Sp(\beta)\rrbracket_M$$
$$\llbracket Sp(\alpha \vee \beta)\rrbracket_M = \llbracket Sp(\alpha)\rrbracket_M \cup \llbracket Sp(\beta)\rrbracket_M$$
$$\llbracket Sp(Sp(\varphi))\rrbracket_M = \llbracket Sp(\varphi)\rrbracket_M$$
$$\llbracket Sp(\neg\varphi)\rrbracket_M = \llbracket Sp(\varphi)\rrbracket_M$$
$$\llbracket Sp(I(\bar{t}))\rrbracket_M = \llbracket Sp(\rho_I(\bar{t}))\rrbracket_M \quad \text{for } I \in \mathcal{I} \text{ with definition } I(\overline{x}) :=_{lfp} \rho_I(\overline{x})$$
$$\llbracket Sp(R(t_1 \ldots, t_n))\rrbracket_M = \bigcup_{i=1}^{n} \llbracket Sp(t_i)\rrbracket_M \text{ for } R \in \mathcal{R}$$
$$\llbracket Sp(f(t_1 \ldots, t_n))\rrbracket_M = \bigcup_{i=1}^{n} \{\llbracket t_i\rrbracket_M\} \cup \bigcup_{i=1}^{n} \llbracket Sp(t_i)\rrbracket_M \text{ if } f \in \mathcal{F}_m$$
$$\llbracket Sp(f(t_1 \ldots, t_n))\rrbracket_M = \bigcup_{i=1}^{n} \llbracket Sp(t_i)\rrbracket_M \text{ if } f \notin \mathcal{F}_m$$
$$\llbracket Sp(ite(\gamma : t_1, t_2))\rrbracket_M = \llbracket Sp(\gamma)\rrbracket_M \cup \begin{cases} \llbracket Sp(t_1)\rrbracket_M \text{ if } M \models \gamma \\ \llbracket Sp(t_2)\rrbracket_M \text{ if } M \not\models \gamma \end{cases}$$
$$\llbracket Sp(ite(\gamma : \alpha, \beta))\rrbracket_M = \llbracket Sp(\gamma)\rrbracket_M \cup \begin{cases} \llbracket Sp(\alpha)\rrbracket_M \text{ if } M \models \gamma \\ \llbracket Sp(\beta)\rrbracket_M \text{ if } M \not\models \gamma \end{cases}$$
$$\llbracket Sp(\exists y\colon y = f(x).\ \varphi)\rrbracket_M = \{\llbracket x\rrbracket_M\} \cup \llbracket Sp(\varphi)\rrbracket_{M[y\leftarrow u]} \quad \text{where } u = \llbracket f(x)\rrbracket_M$$

Fig. 2. Semantics of Support operator. $\llbracket e\rrbracket_M$ refers to the interpretation of an expression e in a model M. The support is defined as the least interpretation satisfying the given equations.

The support of $ite(\gamma : \alpha, \beta)$ always includes the support of γ (since we need to evaluate γ to determine the truth of the formula), but then only adds the support of the case that is evaluated depending on γ. We use the *ite* rather than $\vee$ to write expressions with finer-grained supports in FL. The support of an inductively defined relation term $Sp(I(\overline{x}))$ is simply the support of the body of the definition $Sp(\rho(\overline{x}))$. The body ρ may of course mention I recursively, and the support is the least fixpoint of these equations.

Finally, the support of $\exists y\colon y = f(x).\ \varphi(y)$ contains the location interpreted by x, as well as the support of $\varphi(y)$ where y is interpreted to be location corresponding to $f(x)$ in the given model. Note that although the formula evaluates the same as $\exists y.\, y = f(x) \wedge \varphi(y)$, its support only includes the support of φ for values of y 'matching' the guard, namely $f(x)$.

2.3 Program Verification

We now describe a programming language and the notion of correctness for which we develop automation in this work.

Figure 3 describes the syntax of the language. It supports the typical commands including mutation of fields, allocation, and deallocation of locations. The language also supports function calls, including recursive calls [1].

Operational Semantics. We consider *configurations* of the form (S, H, A) where S is a store, H is a heap, and A denotes the set of allocated locations. Formally, S is a partial map that interprets constants, variables, and non-mutable functions. H is a tuple of maps— one for each mutable function $f \in \mathcal{F}_m$— whose domain is the universe of locations and range is the universe of the appropriate sort (locations for pointers and various background sorts for data fields). We model *nil* as a distinguished location, ensuring in our semantics that valid programs do not dereference *nil*. Note that S and H together define a model where one can interpret FL formulas. We denote the

[1]We do not include *while* loops in the formal syntax to simplify the technical exposition. However, our theory readily extends to programs with iteration, with specifications including frame logic assertions as loop invariants.

$$P ::= x := y \mid x := c \mid v := be \mid x := y.f \mid v := y.d \mid y.f := x \mid y.f := c \mid y.d := v \mid y.d := be$$
$$\mid alloc(x) \mid free(x) \mid \bar{q} := g(\bar{p}) \mid assume(\eta) \mid P;P \mid \text{if } \eta \text{ then } P \text{ else } P \mid return$$

Fig. 3. The syntax of the programming language. Here, x, y are location variables of type Loc, c is a location constant, f is a pointer of type $Loc \to Loc$, d is a data field of type $Loc \to \tau_{\text{bs}}$ for some background sort τ_{bs}, be is a background expression, v is a variable of a background sort, η is a boolean expression without any dereferences. Finally, g is a function of type $\tau_1 \times \cdots \times \tau_m \to \tau'_1 \times \cdots \times \tau'_n$ for some m, n where $\tau_i, \tau'_i \in \mathcal{S}$. This is a method whose body is itself a program whose variables are a superset of the input variables p_i of type τ_i and the output variables q_j of type τ'_j.

satisfaction of an SL formula φ on the model corresponding to a given S, H by $S, H \models \varphi$. A is a subset of the universe of locations that does not contain *nil*. We also introduce an *error* configuration $\perp$.

We describe the operational semantics of various commands manipulating such configurations in Appendix A. The semantics is fairly standard, but we elucidate some key aspects here. First, the operational semantics checks that dereferences are memory safe, i.e., that the lookup or mutation of $x.f$ only occurs when x belongs to the allocated set A (in particular, x is not *nil*). If this does not hold, then the error state $\perp$ is reached. We define $\perp$ to be a sink state, and any command on $\perp$ transitions to $\perp$ itself. Second, allocation adds a *fresh* location (a new element distinct from all the locations in the current universe) to the universe of locations and to the allocated set A. The heap H is also extended, mapping the new location to fixed default values $default_f$ for each $f \in \mathcal{F}_m$. Correspondingly, deallocation removes the deallocated location from A. We do not allow double deallocation, and doing so reaches the error state $\perp$. Finally, functions have call-by-value semantics. We denote a transition between configurations C_1 and C_2 on a program P according to the operational semantics by $C_1 \xrightarrow{P} C_2$.

Hoare Triples and Validity. We consider triples of the form $\{\alpha\}\, P\, \{\beta\}$ where α and β are quantifier-free frame logic formulas and P is a program in the above language. We treat free variables in α and β as constants, implicitly quantifying over them universally. We then define:

Definition 2.1 (Hoare Triple Validity). $\{\alpha\}\, P\, \{\beta\}$ is *valid* if for every configuration (S, H, A) such that $(S, H) \models \alpha$ and $[\![Sp(\alpha)]\!]_{(S,H)} = A$:

(1) (S, H, A) does not transition to $\perp$ on P according to the operational semantics, and
(2) if $(S, H, A) \xrightarrow{P} (S', H', A')$, then $(S', H') \models \beta$ and $[\![Sp(\beta)]\!]_{(S',H')} = A'$

Note that we require the allocated set in the post-state to be *equal* to the support of the postcondition.

Informally, the above definition says that the Hoare Triple is valid if starting from any configuration satisfying the precondition where the allocated set is precisely the support of the precondition, (a) the program does not behave erroneously (e.g., make unsafe dereferences or free twice), and (b) if it reaches a final configuration then the postcondition must hold, with the allocated set in the post state precisely equal to the support of the postcondition.

Relaxed Postconditions. We introduce a modifier *RP* for postconditions, read *relaxed post*, to indicate that the allocated set in the post-state need not be 'tight' for the postcondition. We denote these triples by $\{\alpha\}\, P\, \{RP\colon \beta\}$. Their correctness is defined similarly to that of $\{\alpha\}\, P\, \{\beta\}$ above, except in condition (2) we only require $[\![Sp(\beta)]\!]_{(S',H')} \subseteq A'$.

A One-Way Fragment of FL. In this work we design our VC generation and reasoning mechanisms to ensure that the resulting formulas belong to the L_{oneway} fragment of FORD containing 'one-way' functions. To ensure this we define a fragment of frame logic called FL_{oneway} that only admits one-way functions. Formally, we require in FL_{oneway} that every function symbol in $\mathcal{F}$ of arity n whose range is the foreground sort (Loc) has domain Loc^n. Further, recursively defined symbols in $\mathcal{I}$ of arity k have domain Loc^k. These restrictions do not hamper expressive specifications in practice, and indeed we write specifications for various data structure manipulating programs in FL_{oneway} (see Section 6).

Note that our VC generation (Section 3) as well as reasoning (Section 4) mechanisms are quite general and can be applied to specifications written in the entire FL fragment (Figure 1). However, when specifications are written in FL_{oneway}, the resulting VCs can be converted to formulas in L_{oneway} where the reasoning mechanism we use is *complete* with respect to the FO (i.e., fixpoint) semantics of recursive definitions.

2.4 Eliminating Quantification Using the Cloud Operator

Quantification, restricted in form though it may be, presents a problem for automation. This is especially true when considering quantification that occurs inside inductive definitions. In this work we wish to work over quantifier-free specifications, generating quantifier-free verification conditions in the L_{oneway} fragment. However, one cannot eliminate quantification easily. For example, a definition expressing that x points to a linked list is written in FL as follows [Murali et al. 2023]:

$$List(x) :=_{lfp} ite(x = nil, \top, \exists y : y = next(x).\, List(y) \land x \notin Sp(List(y))) \tag{1}$$

where an existential quantifier is used in the above definition to say that when $x \neq nil$, then y, the "next" location of x must point to a linked list such that x does not belong to the support of $List(y)$[2].

It is tempting to think that the quantifier can be eliminated by simply replacing y with $next(x)$ in the above definition. However, the second conjunct under the quantifier then becomes $x \notin Sp(List(next(x)))$, which does not hold because the support of a formula that mentions $next(x)$ always contains x[3].

In this work we introduce a new operator called the *Cloud* operator, denoted by square brackets $[\cdot]$. The cloud of a formula (resp. term) α is a formula $[\alpha]$ that evaluates the same way as α, but whose support is empty. Simply, $[\alpha]$ treats α as a support-less expression. We extend Frame Logic with the cloud operator, which allows us to rewrite the above definition of *List* without quantification:

$$List(x) :=_{lfp} ite(x = nil,\ \top,\ next(x) = next(x) \land List([next(x)]) \land x \notin Sp(List([next(x)]))) \tag{2}$$

where all occurrences of y are replaced with $[next(x)]$. We also add the tautological conjunct $next(x) = next(x)$ to ensure that the support of the resulting formula still contains $\{x\}$. We capture this transformation formally below.

Formally, we define the semantics of the cloud operator as follows:

$$\begin{array}{llll} Sp([\varphi]) = \emptyset & \text{and} & M \models [\alpha] \text{ iff } M \models \alpha & \text{for a formula } \alpha \\ Sp([t]) = \emptyset & \text{and} & [\![[t]]\!]_M = [\![t]\!]_M & \text{for a term } t \end{array}$$

We then have the following lemma. Fix a model M.

Lemma 2.2 (Eliminating Quantification using the Cloud Operator). *$M \models \exists y\colon y = f(x).\ \varphi$ if and only if $M \models (f(x) = f(x)) \land \varphi([f(x)])$, and $[\![Sp(\exists y : y = f(x).\ \varphi)]\!]_M = [\![Sp((f(x) = f(x)) \land \varphi([f(x)]))]\!]_M$* □

[2]For readers familiar with separation logic, this formula is similar to the SL formula $\exists y.\ (x \overset{next}{\mapsto} y) * List(y)$

[3]The example shows that substitution is not a semantics-preserving operation in FL. Indeed, a substitution is only valid if the replacement expression has the same truth value *and* the same support as the original substituted sub-expression.

We skip the proof of this lemma as it essentially follows from the definition of the support and cloud operators.

In the sequel, we present verification condition generation and automated validity checking for programs annotated with quantifier-free frame logic specifications obtained according to Lemma 2.2. The user writes specifications in Frame Logic with guarded quantification (as in Figure 1), and we rewrite it to quantifier-free formulas involving the cloud operator before applying the procedures described in the following sections.

3 GENERATING VERIFICATION CONDITIONS IN FRAME LOGIC

In this section, we present the technical contribution of the paper, a verification condition generation technique for quantifier-free frame logic specifications.

Let us fix some notation for this section. Let us assume we are dealing with the verification of a program where locations have a fixed set of pointer and data fields $\mathcal{F}_m$, and we let f, f', f_1, etc. range over F. Let us fix a set of recursive definitions for function symbols in a finite set $\mathcal{I}$. Let I, I', I_1, etc. range over $\mathcal{I}$. Let us denote by *Def* the recursive definitions for each $I \in \mathcal{I}$, and let I's definition be $I(\vec{x}) =_{lfp} \rho_I(\vec{x})$. Recursive definitions can be mutually defined.

An annotated program is a triple $\{Pre\}\, P\, \{Post\}$, where P is a program and *Pre*, *Post* are FL formulas over the variables, pointers, data fields, recursive functions and allocated set of P. Let us also fix a set of methods G, which are called by P. We assume every $g \in G$ has its own precondition Pre_g and postcondition $Post_g$.

3.1 Hoare triples over Basic Blocks

Let us fix a Hoare triple for a program: $\{Pre\}\, P\, \{Post\}$ to be verified. We generate a set of Hoare triples over basic blocks to capture the above Hoare triple. These triples not only capture verification of the postcondition when P terminates along different control flows in P but also ensure that all dereferences are safe.

We now define basic blocks corresponding to a program P, $BB(P)$.

- $BB(c) = \{c\}$ for any atomic command c
- $BB(\mathit{if}\ \eta\ \mathit{then}\ P_1\ \mathit{else}\ P_2) = \{assume(\eta); P_1' \mid P_1' \in BB(P_1)\} \cup \{assume(\neg\eta); P_2' \mid P_2' \in BB(P_2)\}$
- $BB(P_1; P_2) = \{P_1'; P_2' \mid P_1' \in BB(P_1), P_2' \in BB(P_2)\}$

$BB(P)$ gives the set of basic blocks of P, splitting paths on conditionals and including the condition in conditionals using assume-statements. Each basic block is assumed to be in "SSA form" (every variable is assigned at most once). This is not a restriction as every basic block in our language can be put in this form.

The set of Hoare triples associated with basic blocks are:

- For each $s \in BB(P)$, $\{Pre\}\, s\, \{Post\}$
- For every s_1, s_2, where $s_1; x := y.f; s_2$ or $s_1; y.f := x; s_2$ or $s_1; \mathit{free}(y); s_2$ is in $BB(P)$,
$$\{Pre\}\, s_1\, \{RP : [y \in A]\}$$
- For every s_1, s_2 such that $s_1; \bar{b} := g(\bar{a}); s_2 \in BB(P)$,
$$\{Pre\}\, s_1\, \{RP : Pre_g[\bar{p} \leftarrow \bar{a}]\}$$

The first item above includes the verification of the postcondition of the program along every basic block. The second set of Hoare triples captures safety of dereferences and ensures freed locations are currently allocated. For each statement that dereferences a location variable or frees a location, we add a Hoare triple for the prefix of the basic block till that statement and check using relaxed postcondition that the variable is in the current allocated set. Finally, for each statement that calls a method g, we introduce a prefix of the basic block till the call and check, again using a relaxed postcondition, that the precondition of the called method holds. The relaxed postcondition above ensures that the heaplet which is the support of the precondition of g, Pre_g (which g will remain within) is a subset of the currently allocated set.

3.2 Verification Condition Generation

We present the core technical contribution in this section, a verification condition generation mechanism for basic blocks. We generate verification conditions in frame logic with only universal quantification over locations (for satisfiability) and with functions mapping from the foreground sort to the foreground or background sorts (integers, sets of locations, etc.).

Intuitively, we will be constructing, for each Hoare triple $\{Pre\}\ s\ \{Post\}$ a verification of the form $((Pre \wedge T) \Rightarrow Post) \wedge SC$ where T captures the semantics of transformation the basic block s has on the state and heap, and SC is a support condition that demands that the support of $\{Post\}$ is the support of Pre modulo allocations and freeing of locations that happen in s (including calls to other functions). In the case of Hoare triples with relaxed postconditions $\{Pre\}\ s\ \{RP : Post\}$, the verification condition is similar, except that the support condition will demand that the support of the postcondition is a subset of the support of the precondition, modulo allocations and freeing of locations.

Consider a Hoare triple $\{\alpha\}\ s\ \{\beta\}$ where s is a basic block and where α, β are annotations in frame logic. We use the notion of a *local configuration* to build the verification condition. We describe a transformation of local configurations across statements of a basic block bb, processing one statement of bb at a time, left to right. A local configuration consists of a 5-tuple (T, A, H, Fr, RD). After processing a prefix s' of a basic block s, T corresponds to the formula describing the state after the transformation s' has had on states satisfying the precondition. The component A represents the set of allocated locations after the prefix, H represents the heap after executing the prefix, and Fr stands for a set of frame rules gathered during mutations of the heap by the prefix. Finally, RD represents a set of new recursive definitions gathered to represent the recursive definitions $\mathcal{I}$ evaluated on intermediate heaps during the execution of the prefix s'.

In the beginning, T is set to Pre and A is set to the support of the precondition α of the Hoare triple, i.e., $Sp(\alpha)$, H is initially simply a set of uninterpreted functions denoting the initial heap, and Fr is the empty set. The set RD is initialized to $\mathcal{D}$, the recursive definitions of $\mathcal{I}$ on the initial heap. Formally:

Definition 3.1. A *logical configuration* is a tuple (T, A, H, Fr, RD) where:

- T is a quantifier-free logical formula describing the program transformation. T is expressed over program variables and recursive functions defined in RD
- A is a term whose type is a set of locations, and denotes the current allocated set of locations.
- H is a map that captures the current heap; it associates with each pointer $f \in F$ a map $H(f) : Loc \longrightarrow \sigma'$, where σ' is the foreground sort of Locations or a background sort. $H(f)$ is expressed as $\lambda u.t(u)$, where t is a term over u and the program variables.
- Fr is a set of universally quantified FO formulae that denotes a set of *frame rules* gathered for each mutation of the heap.
- RD is a set of recursive definitions of a set of function symbols that capture properties/data structures of intermediate heaps. □

The heap map H associates with each pointer $f \in F$, a map $H(f)$, that maps locations to the appropriate sort (either location sort or a background sort). Note that these hence satisfy the one-way condition we want for FO-completeness. The map $H(f)$ is itself expressed in logic using quantifier-free term involving other function symbols, such as the functions that characterize the initial heaplet. For example, $H(f)$ may be the formula $\lambda u.ite(u = z, g_1(z), g_2(u))$, where g_1, g_2 are function symbols and z is a program variable. In general, we will expand the signature with new function symbols as we process a basic block.

We start with a set of recursive definitions for function symbols in $\mathcal{I}$. As we construct the verification condition, we adapt these definitions of functions in $\mathcal{I}$ to ones that refer to them interpreted on new heaps.

In particular, for any H and $I \in \mathcal{I}$, let us introduce a new function symbol I_H, and let us add a recursive definition for I_H. This recursive definition $I[H]$ is the definition for I where every pointer and data field $f \in \mathcal{F}_m$ mentioned in the recursive definition is replaced by $H(f)$. In other words, for each recursively defined function $I \in \mathcal{I}$, where I is defined as $I(x) =_{lfp} \rho(x)$, we give a definition $I[H] : I_H(x) =_{lfp} \rho[H(f)/f](x)$, where $\rho[H(f)/f]$ is ρ where every occurrence of f is replaced with the term $H(f)$, for each $f \in \mathcal{F}_m$.

For example, consider the example of H mentioned above, and consider the definition $lseg(x, y) =_{lfp} (x = y) \vee lseg(f(x), y)$. Then $lseg[H]$ is the definition $lseg_H =_{lfp} (x = y) \vee lseg(ite(x = z, g_1(z), g_2(x)), y)$.

The generation of verification conditions are presented in Figure 4. However, before diving into explaining the rules, let's discuss the frame rule (formally also described in Figure 4) that is used in the rules.

Frame conditions: For any recursively defined function symbol $I(\vec{y}) \in \mathcal{I}$, two heap maps H, H', and a set of locations X, we define the frame rule to be

$$fr(I, X, H, H') \ : \ \forall \vec{y} \in Loc, (X \cap Sp(I_H(\vec{y})) = \emptyset) \implies (I_{H'}(\vec{y}) = I_H(\vec{y}))$$

The above formula is meant to be used when a heap H is transformed to a heap H' where mutations happen only on the locations in X. It says that if the support of the recursive definition on $\vec{y}$ (evaluated in H) does not intersect X, its value in H' is the same as that in H. Note that this formula has only universal quantification over the foreground sort of locations, and hence its translation to FORD will be in the L_{oneway} fragment.

We use $fr(X, H, H')$ to denote the conjunction of $fr(I, X, H, H')$, for each $I \in \mathcal{I}$, which states the frame condition for each recursively defined function in $\mathcal{I}$.

VC Generator:

Definition 3.2 (VC for a basic block). For Hoare triples of basic blocks $\{\alpha\}\, s\, \{\beta\}$ and for basic blocks with relaxed post, the associated verification conditions are defined as:

$$VC(\{\alpha\}\, s\, \{\beta\}) = \left(\left(\bigwedge Fr \wedge T\right) \Rightarrow \beta\right) \wedge Sp(\beta) = A$$

$$VC(\{\alpha\}\, s\, \{RP : \beta\}) = \left(\left(\bigwedge Fr \wedge T\right) \Rightarrow \beta\right) \wedge Sp(\beta) \subseteq A$$

where

$$(T, A, H, Fr, RD) = VC((T_0, A_0, H_0, Fr_0, RD_0), s)$$

where $T_0 = \alpha$, $A_0 = Sp(\alpha)$, $H_0(f) = \lambda u.f(u)$ (for each $f \in F$), $Fr_0 = \emptyset$, $RD_0 = RD$, and VC is the transformer defined formally in Figure 4. □

We now give the intuition behind the transformer VC.

For an assignment x := y (**Assn**), we just add the condition $x = y$ as a conjunct to our transformation formula. Recall that we have assumed that the basic block is in SSA form, and this is the sole assignment to the variable x, and hence this conjunct captures the program configuration constraining x and y to be equal. The scalar assignment rule (**ScalarAssn**) for assigning variables of background sorts is similar.

Assn: VC$((T, A, H, Fr, RD),$ `x := y`$) = (T', A, H, Fr, RD)$, where $T' = T \wedge (x = y)$
ScalarAssn: VC$((T, A, M, Fr, RD),$ `v := be`$) = (T', A, M, Fr, RD)$, where $T' = T \wedge (v = be)$
Deref: VC$((T, A, H, Fr, RD),$ `x := y.f`$) = (T', A, H, Fr, RD)$, where $T' = T \wedge (x = H(f)(y))$
Mutation: VC$((T, A, H, Fr, RD),$ `y.f := x`$) = (T, A, H', Fr', RD')$, where

- $H'(f) = \lambda\ arg.\ ite(arg = y,\ x,\ H(f)(arg)); H'(f') = H(f')$, for every $f' \in \mathcal{F}_m, f' \neq f$
- $Fr' = Fr \cup \{fr(\{y\}, H, H')\}$
- $RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\}$

Alloc: VC$((T, A, H, Fr, RD'),$ `alloc(x)`$) = (T', A', H', Fr, RD)$ where

- $T' = T \wedge (x \notin A)$
- $A' = A \cup \{x\}$
- $H'(f) = \lambda arg.\ ite(arg = x,\ default_f,\ H(f)(arg))$, for every $f \in \mathcal{F}_m$ with domain A'
- $RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\}$

Free: VC$((T, A, H, Fr, RD'),$ `free(x)`$) = (T, A', H, Fr, RD)$ where

- $A' = A \setminus \{x\}$
- $H' = H$ but with domain A'
- $RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\}$

Call: VC$((T, A, H, Fr, RD), \bar{b} := g(\bar{a})) = (T', A', H', Fr', RD')$,
where g has a method with a definition with input parameters $\bar{x}$ and output $\bar{y}$, with precondition $\alpha(\bar{x})$ and postcondition $\beta(\bar{x}, \bar{y})$.
Let α_H denote α where each $f \in \mathcal{F}_m$ is replaced by $H(f)$ and each $I \in \mathcal{I}$ by I_H.
Similarly, let $\beta_{H'}$ denote β where each $f \in \mathcal{F}_m$ is replaced by $H'(f)$ and each $I \in \mathcal{I}$ by $I_{H'}$.
For every $f \in \mathcal{F}_m$, we introduce a fresh uninterpreted function symbol f'. We set

- $H'(f) = \lambda arg.\ ite(arg \notin Sp(\alpha[\bar{x} \leftarrow \bar{a}]),\ H(f)(arg),\ f'(arg))$ with domain A'
- $T' = T \wedge \beta_{H'}[\bar{x} \leftarrow \bar{a}, \bar{y} \leftarrow \bar{b}]$
- $A' = (A \setminus Sp(\alpha_H[\bar{x} \leftarrow \bar{a}])) \cup Sp(\beta_{H'}[\bar{x} \leftarrow \bar{a}, \bar{y} \leftarrow \bar{b}])$
- $Fr' = Fr \cup \{\ fr(Sp(\alpha_H[\bar{x} \leftarrow \bar{a}]), H, H')\ \}$
- $RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\}$

Assume: VC$((T, A, H, Fr, RD), assume(\alpha)) = (T \wedge \alpha, A, H, Fr, RD)$
SeqComp: VC$((T, A, M, Fr, RD), P; Q) =$ VC(VC$((T, A, M, Fr, RD), P)$, Q)
Frame Conditions: In the above rules, the frame formula is defined as:

$$fr(X, H, H') = \bigwedge_{I \in \mathcal{I}} fr(I, X, H, H')$$

Fig. 4. Verification Condition Generation: Predicate transformers for basic blocks

For a dereference `x := y.f` (**Deref**), the transformation looks up the current pointer f, using the function $H(f)$, and adds the conjunct that equates x to $H(f)(y)$. Note that $H(f)(y)$ is the formula $H(f)$ with y substituted as its parameter, and hence results in a quantifier-free term of the appropriate type. The **Assn** and **Deref** rules do not change components other than the first as the heap is not modified.

The generation of VCs for assume statements (**Assume**) and sequential composition (**SeqComp**) are as expected.

For a mutation `y.f := x`, the transformation component doesn't change, but the heap is updated to reflect the mutation, where $H'(f)(y) = x$ and is $H(f)$ on other locations. We also introduce the recursive definitions for $\mathcal{I}$ adjusted for the current heap H' (adding them to RD'). And introduce frame rules $fr(\{y\}, H, H')$ that set these new recursive definitions on any tuple to the value of old ones provided that y does not belong to the support of these definitions on the tuple.

The transformation for function calls is more complex. First, we introduce an elegant way to take care of the heap transformation (as far as we know, this is novel). For each pointer and data field $f \in \mathcal{F}_m$, we introduce a *fresh* function symbol f'. The map $H'(f)$ is now an *ite* (if-then-else) term, where $H'(f)(u)$ evaluates to $H(f)(u)$, which is the old value of the pointer f, provided u does not belong to the support of the precondition of the called function g. Otherwise, $H(f)(u)$ evaluates to $f'(u)$, which is an arbitrary value since f' is uninterpreted. The transformation component imbibes the postcondition guaranteed by the call to g, where the recursively defined functions R in the postcondition are modified to the functions $R_{H'}$. Recursive definitions are adapted to the new heap H' after the call, and added to RD'. The new allocated set is derived from the old allocated set by removing the locations in the support of the precondition of g and adding back the locations in the support of the postcondition of g. This handles the locations g may allocated or freed during its execution. Finally, we add frame condition formulae $fr(Sp(\alpha[\bar{x} \leftarrow \bar{a}]), M, M')$ that says that for any recursive definition on a tuple that has an empty intersection with the support of the precondition of g remains unaffected when evaluated in the new heap.

The transformation for `alloc` statement (**Alloc**) adds an assumption that the allocated location is not already present in the current allocated set A, adds it to the new allocated set, and updates the heap map so that the new location's pointers point to default values. The transformation for `free` statements (**Free**) simply removes the location x from the allocated set. Even though logical expression for the heap function H does not change with freeing of locations, *its domain does*. Hence, when either allocation and free happens, we recompute the recursive definitions on the new heap by updating the RD component. Note that we do not check whether freed locations are indeed allocated as those are checked on other basic blocks explicitly, as described in the previous subsection.

Our VC generation is sound and complete for the validity of Hoare Triples over basic blocks. Fix a triple $\{\alpha\}\, s\, \{\beta\}$ where s is a basic block, and let VC be the map defined in Definition 3.2. Then:

Theorem 3.3 (Soundness of VC Generation for Basic Blocks). *If $VC(\{\alpha\}\, s\, \{\beta\})$ is a valid formula in Frame Logic then $\{\alpha\}\, s\, \{\beta\}$ is valid in the sense of Definition 2.1. Similarly, if $VC(\{\alpha\}\, s\, \{RP\colon \beta\})$ is valid in Frame Logic then the triple $\{\alpha\}\, s\, \{RP\colon \beta\})$ is valid.*

Theorem 3.4 (Completeness of VC Generation for Call-Free Basic Blocks). *Let s be a basic block with no function calls. If $\{\alpha\}\, s\, \{\beta\}$ is valid (Definition 2.1) then $VC(\{\alpha\}\, s\, \{\beta\})$ is valid. Similarly, if $\{\alpha\}\, s\, \{RP\colon \beta\})$ is valid then $VC(\{\alpha\}\, s\, \{RP\colon \beta\})$ is valid.*

We elide the proofs of these theorems as they are fairly straightforward from the definition of the VC transformer and the operational semantics (Appendix A). The key argument in the proof of completeness is that we model mutations *precisely* using updates to the heap map H. Similarly, allocation and deallocation are also modeled precisely using the symbolic allocated set A, which corresponds to the true allocated set as defined by the operational semantics for call-free blocks. Our completeness result is interesting in its own right, and showcases the power of our framework. Contemporary works on VC generation for other logics such as Separation Logic are often not complete [Berdine et al. 2005].

4 VALIDATING VERIFICATION CONDITIONS

In this section, we describe our validity procedure for checking the quantifier-free frame logic verification conditions generated by the mechanism described in Section 3. Our technique has two stages. In the first stage, we translate the quantifier-free FL formulas to quantifier-free FORD formulas in a way that preserves validity. Recall that quantifier-free FL is simply an extension of quantifier-free FORD with the support $Sp(\cdot)$ and cloud $[\cdot]$ operators. Intuitively, our translation encodes the semantics of these operators in FORD itself. In the second stage, we reason with the quantifier-free FORD formulas using natural proofs, a mechanism developed in prior work [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013].

4.1 Stage 1: Translating Quantifier-Free FL to Quantifier-Free FORD

The first stage of our reasoning mechanism translates FL formulas to FORD formulas. The key idea is to encode the semantics of the Sp and $[\cdot]$ operators within FORD itself. This process introduces new recursive definitions corresponding to the support of existing recursive definitions.

Let us denote the translation by Π. This function takes as input a quantifier-free FL formula (or term) containing the Sp and $[\cdot]$ operators and outputs an FORD formula (resp. term). We describe the formal translation in Appendix B and provide intuition here. First, we obviously have that for any FL formula α that does not mention the support or cloud operators, $\Pi(\alpha) = \alpha$.

Next, to encode the Sp operator, it turns out that we can essentially mimic the equations describing the semantics of Sp (Figure 2)! This is easy to see for formulas that do not contain recursively defined symbols. For example, $\Pi(Sp(f(x) = y)) = \{x\}$ for a mutable function $f \in \mathcal{F}_m$, since $\Pi(f(x) = y) = \Pi(f(x)) \cup \Pi(y) = \{x\} \cup \emptyset$. Similarly, $\Pi(Sp(\alpha \wedge \beta)) = \Pi(Sp(\alpha)) \cup \Pi(Sp(\beta))$. In this way, the Sp operator can be 'compiled away' by simply following the equations in Figure 2.

However, this does not work for recursively defined symbols. For example, recall the definition of $List(x)$ written using the cloud operator (Equation 2). Using the recipe described above, the support of $List(x)$ yields the following equation:

$$\begin{aligned}Sp(List(x)) &= Sp(ite(x = nil,\ \top,\ next(x) = next(x) \wedge List([next(x)]) \wedge x \notin Sp(List([next(x)])))) \\ &= ite(x = nil, \emptyset, \{x\} \cup Sp(List([next(x)]))\end{aligned}$$

where we have expanded Sp on a cloud expression using the equation $Sp([\alpha]) = \emptyset$.

We cannot compile away the Sp operator in this case. However, recall that the Sp operator is the least interpretation satisfying the equations in Figure 2. We therefore introduce a new recursively defined symbol Sp_{List}, defined by

$$Sp_{List}(x) :=_{lfp} ite(x = nil, \emptyset, \{x\} \cup Sp_{List}([next(x)]))$$

and we then translate $Sp(List(x))$ to $Sp_{List}(x)$. The latter is now an expression in FORD mentioning the newly created recursive definition. In general, our translation creates a new recursively defined symbol Sp_I for every $I \in \mathcal{I}$ corresponding to its support.

Finally, the encoding of the $[\cdot]$ operator is trivial. Since support expressions were eliminated in the previous step, $[\alpha]$ is identical to α. Therefore, we simply ignore it in the translation. In particular, the recursive sub-expression $Sp_{List}([next(x)])$ in the above definition becomes $Sp_{List}(next(x))$ in the final translation to FORD.

4.2 Stage 2: Reasoning with FORD Formulas using Natural Proofs and SMT Solvers

The above translation of frame logic formulas results in FORD formulas that are quantifier-free (i.e., all free variables are implicitly universally quantified). Furthermore, they have the special form that functions from the location sort map to either the location sort or a background sort, but no function maps background sorts to the location sort. Hence they are part of a special fragment with "one-way" definitions, $\mathcal{L}_{oneway}$ fragment.

Natural proofs [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013] are a sound but incomplete technique for proving validity of FORD formulae. It works by treating recursive definitions to have fixpoint semantics (rather than least fixpoint) and hence obtaining a first-order formula. It then instantiates the recursive definitions (similar to instantiating the quantifier for these definitions) on terms of depth d (for larger and larger d), and obtaining quantifier-free formulas. Validating

$$\begin{array}{rrcl} \textit{SL-FL}_b \text{ Formulas} & \alpha, \beta & := & \gamma \mid x \xrightarrow{f} y \mid \alpha \wedge \beta \mid \alpha * \beta \mid \alpha \vee \beta \mid ite(\gamma, \alpha, \beta) \mid \exists y.(x \xrightarrow{f} y : \alpha) \\ \text{H.I. atomic formulas} & \gamma & := & true \mid false \mid x = y \mid x \neq y \mid x = nil \mid x \neq nil \end{array}$$

Fig. 5. Base Separation Logic with FL inspired semantics (*SL-FL*$_b$); H.I. stands for "heap-independent"

these quantifier-free formulas is performed using an SMT solver. The foundations of this technique [Löding et al. 2018] show that the technique is in fact a complete technique for dealing with first-order formulae (where recursive definitions have fixpoint semantics). Furthermore, in practice, natural proofs have been shown useful for heap verification of other logics. We can hence validate verification conditions using this automated technique.

5 A SEPARATION LOGIC WITH FRAME LOGIC INSPIRED SEMANTICS

We now define a separation logic with *alternate semantics* guided by frame logic semantics, in particular defining determined supports for expressions that do not depend on the truthhood of the formula. We also show a translation of this separation logic to frame logic, where tight heaplets in the former translate to supports in the latter. This allows us to use the automated verification techniques presented in the previous sections to reason with programs annotated with this separation logic. Our logic is powerful, subsuming several known precise separation logics in the literature [Berdine et al. 2005; Murali et al. 2023; O'Hearn et al. 2004]. In particular, our logic allows negation and disjunction for spatial formulae which are not supported by prior precise separation logics. Our logic, however, defines the semantics of disjunction differently in order to ensure unique heaplets.

We define first a base separation logic (*SL-FL*$_b$) where the semantic differences are clear. We then extend it to a more expressive logic (*SL-FL*) with inductive definitions and background sorts. Both these logics can be translated to FL and hence verification of programs annotated with these logics can be automated using the results of the previous section.

5.1 Base logic *SL-FL*$_b$

Let us fix a set of locations Loc. Let $Loc?$ denote $Loc \cup \{nil\}$ where nil is a special symbol and $nil \notin Loc$. Let us fix a countable set of variables Var, and let x, y, x_1, x_2, x', y' etc. range over Var. These variables will be used to intuitively model program variables as well as quantified variables. Let us fix a finite set of pointers Ptr, and let f, f', g, etc. range over Ptr.

The syntax for the logic *SL-FL*$_b$ is given in Figure 5. Here, x, y range over Var and $f \in Ptr$. The logic supports the separating conjunction $*$, both disjunction and conjunction, a special if-then-else construct *ite*, and guarded existential quantification. Guards in *ite* formulas are restricted to be atomic guards that have no dereferences (we will relax this later when extending the logic).

Semantics:. A *store* is a function $s : Var \rightarrow Loc?$.

A *heaplet* h is a set of functions $\{h_f \mid f \in Ptr\}$, where each $h_f : H \rightarrow Loc?$, where $H \subseteq Loc$ is the common domain for all the functions h_f. Note that we do not assume heaplets are finite.[4] Note that the domain of all the functions h_f is common. We denote by $h(f)$ the function h_f that h defines, and by $dom(h)$ the set of locations H. Also, for any $S \subseteq dom(h)$, let $h \downharpoonright S$ denote the heaplet where the domains of the pointer fields in h are restricted to S. A heaplet h' is a subheap/subheaplet of h if there is some $S \subseteq dom(h)$ such that $h' = h \downharpoonright S$. A global heap is a heaplet h with $dom(h) = Loc$.

[4]In our logics, heaps can be infinite, and so can heaplets, like the heaplet of an infinite list.

$$
\begin{array}{ll}
\mathit{Supp}(\gamma, s, h) & = \emptyset,\ \textit{for any heap-independent atomic formula } \gamma \\
\mathit{Supp}(x \xrightarrow{f} y, s, h) & = \{s(x)\}\ \textit{if } s(x) \textit{ is in } \mathit{dom}(h) \textit{ and } \bot \textit{ otherwise} \\
\mathit{Supp}(\alpha \oplus \beta, s, h) & = \mathit{Supp}(\alpha, s, h) \cup \mathit{Supp}(\beta, s, h),\ \textit{where } \oplus \in \{\wedge, *, \vee\} \\
\mathit{Supp}(\mathit{ite}(\gamma, \alpha, \beta), s, h) & = \mathit{Supp}(\alpha, s, h)\ \textit{if } s \models \gamma \textit{ and } \mathit{Supp}(\beta, s, h) \textit{ otherwise} \\
\mathit{Supp}(\exists y.(x \xrightarrow{f} y : \alpha), s, h) & = \bot \textit{ if } s(x) \notin \mathit{dom}(h), \textit{and} \{s(x)\} \cup \mathit{Supp}(\alpha, s[y \mapsto h(f)(s(x))], h) \textit{ otherwise.}
\end{array}
$$

Fig. 6. Definition of supports for separation logic formulae with respect to a heaplet

$$
\begin{array}{llll}
(s,h) & \models & \gamma & \textit{iff} \quad h = \emptyset \textit{ and } s \models \gamma,\ \textit{for any heap-independent formula } \gamma \\
(s,h) & \models & x \xrightarrow{f} y & \textit{iff} \quad h = \{s(x)\} \textit{ and } h(f)(s(x)) = y \\
(s,h) & \models & \alpha * \beta & \textit{iff} \quad \textit{there exists } h_1, h_2 \textit{ subheaplets of } h, \mathit{dom}(h_1) \cup \mathit{dom}(h_2) = \mathit{dom}(h), \\
 & & & \qquad \mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset, (s, h_1) \models \alpha \textit{ and } (s, h_2) \models \beta \\
(s,h) & \models & \alpha \wedge \beta & \textit{iff} \quad (s,h) \models \alpha \textit{ and } (s,h)| = \beta \\
(s,h) & \models & \alpha \vee \beta & \textit{iff} \quad \textit{there exists } h_1, h_2 \textit{ subheaplets of } h, \mathit{dom}(h_1) \cup \mathit{dom}(h_2) = \mathit{dom}(h), \\
 & & & \qquad \mathit{dom}(h_1) = \mathit{Supp}(\alpha, s, h_1), \mathit{dom}(h_2) = \mathit{Supp}(\beta, s, h_2), \textit{and} \\
 & & & \qquad ((s, h_1) \models \alpha \textit{ or } (s, h_2) \models \beta) \\
(s,h) & \models & \mathit{ite}(\gamma, \alpha, \beta) & \textit{iff} \quad (s \models \gamma \textit{ and } (s,h) \models \alpha) \textit{ or } (s \not\models \gamma \textit{ and } (s,h) \models \beta) \\
(s,h) & \models & (\exists y. x \xrightarrow{f} y : \alpha) & \textit{iff} \quad x \in \mathit{dom}(h) \textit{ and } (s[y \mapsto h(f)(x)], h) \models \alpha
\end{array}
$$

Fig. 7. Semantics of base logic $SL\text{-}FL_b$

The Support Map:. Inspired by frame logic, we define a map, called the support map, that maps any formula, store, and heaplet to the *subdomain* of the heaplet that corresponds to the set of locations that the formula's truthhood depends upon.

The map *Supp* is defined in Figure 6. Intuitively, $Supp(\alpha, s, h)$ either maps to a subset $H \subseteq dom(h)$ of locations that is sufficient to determine the truthhood of α, or to $\bot$ (when h is not large enough). This support corresponds roughly to the tightest heaplet that is typically used in separation logic semantics. However, there are some crucial differences. First, note that the support of heap-independent atomic formulas that depend only on the store (like *true*, $x = y$, etc.) is the empty set rather than *any heaplet* as in separation logic. While the supports of formulas $\alpha \wedge \beta$, and $\alpha * \beta$ are similar to tight heaplets defined in separation logic, the rule for disjunctions $\alpha \vee \beta$ is starkly different— it is the union of the supports for α and β, rather than *either the support of* α *or* β, depending on which holds, as in separation logic. This is crucial and makes supports unique, for example when both disjuncts hold. Note that *if-then-else* expressions allow the support to depend on the heap, namely how the condition γ evaluates on the heaplet, evaluating to either the support of α or the support of β. This does not destroy heaplets from being uniquely determined.

The supports of formulas have several properties worth noting. Let S be the support of a formula with respect to a store and a heaplet h, with $S \neq \bot$. First, S will be a subset of $dom(h)$. Second, consider a heaplet h' that agrees with h on S. Then the support of the formula with respect to h' will be S as well. Third, the support of the formula with respect to h restricted to S will be S itself. Finally, there is at most one subheaplet h' of h such that the support of the formula in h' is $dom(h')$ itself. The following lemma formalizes this (see Appendix C for a proof gist).

Lemma 5.1. *Let s be a store and α be a $SL\text{-}FL_b$ formula, h be a heaplet, and let $S = Supp(\alpha, s, h)$, and $S \neq \bot$.*

(1) $S \subseteq dom(h)$.

$$\Pi(\gamma) = \gamma,\ \textit{for H.I. atomic formula } \gamma \qquad \Pi(x \xrightarrow{f} y) = f(x) = y$$
$$\Pi(\alpha * \beta) = \Pi(\alpha) \wedge \Pi(\beta) \wedge Sp(\Pi(\alpha)) \cap Sp(\Pi(\beta)) = \emptyset \qquad \Pi(\alpha \vee \beta) = \Pi(\alpha) \vee \Pi(\beta)$$
$$\Pi(\alpha \wedge \beta) = \Pi(\alpha) \wedge \Pi(\beta) \wedge Sp(\Pi(\alpha)) = Sp(\Pi(\beta)) \qquad \Pi(ite(\gamma, \alpha, \beta)) = ite(\gamma, \Pi(\alpha), \Pi(\beta))$$
$$\Pi(\exists y.(x \xrightarrow{f} y : \alpha)) = \exists y \colon y = f(x).\, \Pi(\alpha)$$

Fig. 8. Translation from *SL-FL*$_b$to Frame Logic

(2) *Let h' be a heaplet such that $S \subseteq dom(h')$ and $h' \downharpoonright S = h \downharpoonright S$. Then $Supp(\alpha, s, h') = S$.*
(3) *$Supp(\alpha, s, h \downharpoonright S) = S$.*
(4) *Let h_1 and h_2 be two sub-heaplets of h and assume $Supp(\alpha, s, h_1) = dom(h_1)$ and $Supp(\alpha, s, h_2) = dom(h_2)$. Then $dom(h_1) = dom(h_2)$.* □

We are now ready to define the semantics of separation logic with respect to a store and a heaplet. The semantics is defined in Figure 7.

The semantics for heap-independent formulas requires heaplets to be empty. The semantics of $x \xrightarrow{f} y$, $\alpha * \beta$, and $\alpha \wedge \beta$ are similar to standard separation logic semantics[Reynolds 2002]. The semantics of disjunctive formulas $\alpha \vee \beta$ is different and uses the *Supp* map. The formula $\alpha \vee \beta$ holds in a heaplet h if the heaplet is the union of the supports of α and β, and α holds with respect to its support or β holds with respect to its support. The semantics of if-then-else (*ite*) formulas ensures that the heaplet matches the support required by α or that of β, depending on whether γ evaluates to true or false, respectively.

The above semantics ensures a crucial property— consider any store s, heap h, and formula α, then there is *at most* one sub-heaplet h' of h that satisfies α. This property does not hold for standard separation logic semantics (see Appendix C for a proof gist).

Lemma 5.2. *Let s be a store and h a heaplet.*
(1) *If $(s, h) \models \alpha$, then $Supp(s, h, \alpha)$ = $dom(h)$.*
(2) *There is at most one subheaplet h' of h such that $(s, h') \models \alpha$.* □

Translation to Frame Logic.
SL-FL$_b$ formulas, with their semantics using determined heaplets/supports, can be readily translated to frame logic. The translation is given in Figure 8, and is simple and natural. The translation preserves both truthhood as well as heaplet semantics in the following sense (proof in Appendix C):

Lemma 5.3. *Let g be a global heap (a heaplet with domain Loc). Let α be an SL-FL$_b$ formula. Then*
- *$g \models \Pi(\alpha)$ iff there exists a heaplet h of g such that $(s, h) \models \alpha$.*
- *If $g \models \Pi(\alpha)$, then $Supp(\alpha, s, g)$ is equal to the value of $Sp(\Pi(\alpha))$ in g.* □

The above allows us to build automatic verification procedures for programs annotated with SL-FL$_b$ formulas. Annotations with contracts using separation logic formulas can be translated to Frame Logic, with the understanding that they define implicit heaplets which is the support of the corresponding Frame Logic formula. We can hence use our automated verification of programs with FL annotations to verify programs with *SL-FL*$_b$ annotations.

5.2 Extending the base logic to background sorts and recursive definitions

We now extend the base logic to a more powerful logic that (a) allows recursive definitions of predicates and functions, (b) incorporates background sorts (like arithmetic), pointer fields to such sorts, and recursively defined functions that evaluate to the background sort and (c) generalizes guards to be arbitrary separation logic formulae.

$$\begin{array}{rrcl}
\text{SL-FL Formulas} & \alpha, \alpha', \beta & := & \mathit{true} \mid \mathit{false} \mid x = y \mid x \neq y \mid x = \mathit{nil} \mid x \neq \mathit{nil} \mid x \xrightarrow{f} y \\
 & & & \mid p(\bar{t}) \mid \alpha \wedge \beta \mid \alpha * \beta \mid \alpha \vee \beta \mid \mathit{ite}(\alpha', \alpha, \beta) \mid \exists y.(x \xrightarrow{f} y : \alpha) \mid R(\overline{x}) \\
\text{Terms} & t, t', t_i & := & x \mid t.f \mid g(\bar{t}) \mid \mathit{ite}(\alpha, t, t') \mid F(\bar{t}) \\
\textit{Recursive Definitions:} & R(\overline{x}) & :=_{lfp} & \rho_R(\overline{x}, \mathcal{R}, \mathcal{F}),\ \textit{for each } R \in \mathcal{R} \\
 & F(\overline{x}) & :=_{lfp} & \mu_F(\overline{x}, \mathcal{R}, \mathcal{F}),\ \textit{for each } F \in \mathcal{F}
\end{array}$$

Fig. 9. *SL-FL*: Syntax of full Separation Logic with Frame-Logic inspired semantics. The guards in *ite* expressions should not mention recursively defined relations $R \in \mathcal{R}$

Let us fix background sorts, with accompanying functions and relations, $\mathcal{G} = \{g_1, \ldots, \}$ and $\mathcal{P} = \{p_1, \ldots, \}$. Let us also fix a set of symbols for functions and relations, $\mathcal{F}$ and $\mathcal{R}$, respectively, which we will use to define new *recursively-defined* functions and relations. Heaplets are extended to include pointers that map locations to background sorts.

The syntax of *SL-FL* is given in Figure 9. Syntax for *SL-FL* formulas is similar to base logic, but we add the ability for guards to be *SL-FL* formulas themselves (in *ite* expressions), and allow them to evaluate predicates over terms of background sorts ($p(\bar{t})$), and allow evaluation of recursively defined predicates ($R(\bar{t}, \overline{x})$). The syntax for terms allows dereferencing locations (resulting in either locations or background sort), computing functions over background sorts (like + over integers), if-then-else constructs, or recursively defined functions that return values of background sort (like "Keys" of a location x pointing to a list).

Recursively defined predicates (R) and functions (F) are defined with parameters of type location *only*. This is a crucial restriction as we will, upon translation to Frame Logic and later to first-order logic, treat these definitions as universally quantified equations, and it's important for FO-completeness using natural proofs that these quantifications are over the foreground sort of locations only. Definitions of predicates (R) and functions F are mutually recursive and are arbitrary *SL-FL* formulae ρ and terms over background sorts μ that can mention $\mathcal{R}$ and $\mathcal{F}$.

The semantics of the logic extend that of the base logic in the natural way, and we skip the formal semantics here (see Appendix D for some details). We assume that each background sort is a complete lattice (for flat sorts like arithmetic, we can introduce a bottom element and a top element to obtain a complete lattice). The support maps are extended— they include the support of guards in *ite* formulas, and for recursive definitions, they evaluate to the support of their definitions. The semantics of both *Supp* and $\models$ are taken together as equations and their semantics is defined as the least fixpoint of these equations.

The separation logic that emerges is powerful and can state properties such as standard datastructures (lists, trees) and properties of them (keys, bst), etc. Furthermore, the logic can be translated easily to FL (extending the Π translation above) for automated verification of programs annotated using *SL-FL* (see Appendix D for some details).

6 IMPLEMENTATION AND EVALUATION

6.1 Implementation

We developed the Frame Logic Verifier (FLV) tool based on the mechanism described in Sections 3 and 4. The tool allows users to write heap-manipulating programs and annotations in quantifier-free Frame Logic, using the support $Sp(\cdot)$ and cloud $[\cdot]$ operators. The tool also allows users to write inductive lemmas which are then proven automatically using a form of induction using pre-fixpoints [Löding et al. 2018].

The FLV tool generates VCs that are translated to FORD. These quantifier-free FORD formulas are then checked using an existing solver for natural proofs [Murali et al. 2022] which implements the procedure described in Section 4.2, instantiating recursive definitions and reasoning with the resulting quantifier-free formulas using Z3 [De Moura and Bjørner 2008].

The implementation is also optimized, introducing new recursive definitions and frame rules only at certain key points: across function calls or at the end of a basic block. Note that the formulation in Section 3 introduces them across every mutation, allocation, deallocation, and function call (as described in Section 3). These optimized frame rules 'batch' the individual mutations, catering to the *footprint* of the program between the key points. This optimization reduces the number of instantiations produced by natural proofs significantly, leading to effective reasoning.

6.2 Benchmark Suite

We evaluate the expressiveness and performance of our contributions on a suite of data structure programs which we annotate with Frame Logic specifications[5]. Our suite consists of 29 programs involving operations on data structures such as singly and doubly linked lists, circular lists, binary search trees, red-black trees, treaps, etc. We obtained this suite by distilling a core set of benchmarks from prior work [Pek et al. 2014]. We considered the various crucial data structure manipulation algorithms for each structure such as insertion, deletion, search, sorting, and traversals.

Table 1 summarizes our benchmark suite. We report in addition to lines of code some statistics on the annotation burden, split across data structure definitions, specifications (pre/post conditions), and inductive lemmas. We present the number of lines corresponding to recursive definitions and inductive lemmas grouped by data structures as they only need to be written once and can be reused across the various routines.

6.2.1 Specifications. We annotate our programs with complete functional specifications. We define data structures and various measures over them using recursive definitions (see Appendix E for examples) and write specifications using them. Table 2 contains examples of pre and post conditions for several routines. We use the *Sp* operator to express disjointness of data structures; for example, we specify in the precondition for the routine appending two singly linked lists x and y that they are disjoint using the formula $List(x) \wedge List(y) \wedge Sp(List(x)) \cap Sp(List(y)) = \emptyset$.

6.2.2 Lemmas. We use natural proofs to check the translated FORD formulas, which is complete when recursive definitions are interpreted as fixpoints (rather than least fixpoints). Of course, fixpoints are an incomplete abstraction, and therefore it is possible that valid theorems in FORD are not provable using natural proofs. Work on natural proofs bridges this gap by asking the user to provide inductive lemmas [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013].

We follow this paradigm in our work, writing lemmas when natural proofs was unable to prove the programs correct. We summarize the lemmas we wrote in Table 3, which were predominantly of three types. The first type of lemma involves supports of recursive definitions, stating for instance that if x pointed to a binary search tree (BST) as well as a red-black tree (RBT), then the supports of the two definitions are equal. Lemmas of this type also express disjointness of supports.

The other two classes of lemmas relate the set of keys stored in a data structure to the minimum and maximum elements of that set. For example, we state that a key k that is greater than the maximum of a BST cannot be a key stored in that BST. Note that min and max are recursively defined functions. Lemmas of this kind were used to verify search routines, where the programs may choose not to explore a certain subtree (e.g., BST Find) or even terminate search early and

[5]We provide the benchmarks in an anonymized repository: https://github.com/plresearcher/flvbenchmarks

Data Structure	Operations	Program	Rec. Defs.	Lemmas	Pre/Post Conds.
	Insert Front	6			2
	Insert Back	15			2
	Delete	18			2
Singly Linked List	Find	16	5	1	3
	Copy	14			4
	Append	12			3
	Reverse	14			5
	Insert Front	17			3
Doubly Linked List	Insert Back	18	14	2	3
	Insert Middle	19			10
	Delete Middle	15			9
	Insert	22			3
Sorted List	Delete	18	7	5	3
	Find	22			3
	Insertion Sort	30			5
Sorting	Merge Sort	67	12	11	12
	Quicksort	63			45
	Insert Front	15			3
Circular List	Delete Front	27	11	2	10
	Find	32			6
	Insert	31			4
BST	Delete	50	33	5	8
	Find	23			3
	Rotate Left	7			4
	Tree to List	21	31	9	13
Tree	In-order Traversal	14	28	6	6
Treap	Delete	62	46	9	14
	Find	23			7
RBT	Insert	72	46	8	18

Table 1. List of Benchmarks and Lines of Code in the Specifications of Each Component. Each line of code has approximately 80 characters.

Benchmark	Pre Condition	Post Condition
SLL Reverse	$List(x)$	$List(ret) \wedge Keys(ret) = Old(Keys(x))$
Sorted Insert	$Sorted(x) \wedge (k < +\infty)$	$Sorted(ret) \wedge Keys(ret) = Old(Keys(x)) \cup \{k\}$ $\wedge\ ite(Old(Min(x)) < k, Min(ret) = Old(Min(x)), Min(ret) = k)$
Sorted Concat (part of Quicksort)	$Sorted(x) \wedge Sorted(y)$ $\wedge\ Max(x) \leq Min(y)$ $\wedge\ Sp(Sorted(x))$ $\cap\ Sp(Sorted(y)) = \emptyset$	$Sorted(ret) \wedge Keys(ret) = Old(Keys(x)) \cup Old(Keys(y))$ $\wedge\ ite(x = nil, Min(ret) = Old(Min(y)), Min(ret) = Old(Min(x)))$ $\wedge\ ite(y = nil, Max(ret) = Old(Max(x)), Max(ret) = Old(Max(y)))$
Circular Find	$Circ(x)$	$Circ(x) \wedge Keys(x) = Old(Keys(x)) \wedge (ret \iff k \in Keys(x))$
BST Insert	$BST(x) \wedge (-\infty < k < +\infty)$	$BST(ret) \wedge Keys(ret) = Old(Keys(x)) \cup \{k\}$ $\wedge\ ite(k < Old(Min(x)), Min(ret) = k, Min(ret) = Old(Min(x)))$ $\wedge\ ite(k > Old(Max(x)), Max(ret) = k, Max(ret) = Old(Max(x)))$
Treap Delete	$Treap(x)$	$Treap(ret) \wedge Keys(ret) = Old(Keys(x)) \setminus \{k\}$ $\wedge\ Priorities(ret) \subseteq Old(Priorities(x)) \wedge Old(MinKey(x)) \leq MinKey(ret)$ $\wedge\ MaxKey(ret) \leq Old(MaxKey(x)) \wedge MaxPrio(ret) \leq Old(MaxPrio(x))$

Table 2. Example Pre and Post Conditions. x and y are of type Loc, and k is of type Int. ret is a pointer to the data structure returned by the routines.

declare that a key was not found. We believe that future work can automate these classes of lemmas and remove the need for user help altogether.

6.3 Evaluation

We evaluate our tool on our suite of benchmarks. The experiments were performed on a laptop with the following specifications: 11th Gen Intel Core i7-1165G7 @ 2.80GHz, 2701 Mhz, 4 Cores, 8 Logical Processors, and 12 GB RAM.

Lemma Type	Benchmark	Lemmas
Support	BST Find	$BST(x) \implies Sp(Keys(x)) \subseteq Sp(BST(x))$
	RBT Insert	$RBT(x) \implies Sp(BST(x)) = Sp(RBT(x))$
	RBT Insert	$RBT(x) \implies x \neq nil \implies x \notin Sp(RBT([left(x)]))$
Keys	Sorted Find	$Sorted(x) \implies k < Min(x) \implies k \notin Keys(x)$
	BST Find	$BST(x) \implies Max(x) < k \implies k \notin Keys(x)$
Min/Max	Tree to List	$BST(x) \implies n \neq nil \implies$ $ite(left(x) = nil, Min(x) = key(x), Min(x) = Min(left(x)))$
	Treap Delete	$Treap(x) \implies x \neq nil \implies MaxPriority(x) = prio(x)$
	Quicksort	$x \neq nil \implies Max(x) \leq Min(y) \implies Min(x) \leq Min(y)$

Table 3. Example Lemmas

Benchmark	Number of VCs	Verif. Time
SLL Insert Front	3	1s
SLL Insert Back	7	1s
SLL Delete	9	1s
SLL Find	5	1s
SLL Copy	6	1s
SLL Append	5	1s
SLL Reverse	5	1s
DLL Insert Front	8	1s
DLL Insert Back	9	8s
DLL Insert Mid	10	8s
DLL Delete Mid	11	4s
Sorted Insert	10	1s
Sorted Delete	9	2s
Sorted Find	6	1s

Benchmark	Number of VCs	Verif. Time
Insertion Sort	13	9s
Merge Sort	26	12s
Quicksort	25	38s
CL Insert Front	6	29s
CL Delete Front	12	20s
CL Find	10	2s
BST Insert	13	1m20s
BST Delete	27	1m48s
BST Find	8	15s
BST Rotate Left	5	2s
BST to List	11	27s
In-Order Traversal	8	11s
Treap Delete	32	1m08s
Treap Find	8	13s
RBT Insert	39	8m30s

Table 4. Tool Performance (SLL = singly-linked list, DLL = doubly-linked list, CL = circular list, BST = Binary Search Tree).

6.3.1 Verifying Programs. We present our evaluation in Table 4. Our tool verifies all benchmarks effectively. For the more complex programs *Treap Delete* and *RBT Insert*, we executed two instantiation strategies, one on depth 2 terms and the other on an extended footprint (see [Pek et al. 2014]), in parallel, and report the time when a strategy first succeeds. We observe that verification times increase as specifications get more complex, particularly involving reasoning with sets. However, we manually analyzed such benchmarks and found that the reasoning patterns involving sets are quite general and are repeated across the suite. We believe that future work on a paradigm that reasons with sets using set axioms (as in tools like Dafny) can improve performance greatly.

6.3.2 Identifying Buggy Programs. We study the performance of our tool on identifying buggy programs by ablating specifications for five programs. We removed essential conjuncts from the precondition such as the disjointness of two lists (for the sorted list merge routine), or the requirement for the root of a BST to not be *nil* (for deletion in a BST). These ablations were derived from the kinds of incorrect specifications we encountered during the creation of the benchmark suite. Among the invalid VCs generated, the tool is able to report them as unprovable within a few seconds for half of all VCs, but times out in 20 minutes for the other half. We provide further details regarding the ablations and the VCs corresponding to the timeouts in Appendix E.

6.3.3 Verification Experience. The benchmark suite was developed in its entirety by one of the authors. We report on the insights from their experience in creating the benchmark suite, writing specifications in Frame Logic, and verifying programs using the FLV tool. This author was not familiar with Frame Logic or Natural Proofs before beginning this work and was not involved in the development of the FLV tool. They were also not aware of the details of the VC generation mechanism during the development of the suite. They were only familiar with First-Order Logic and general principles of program verification and had some experience using Coq. We do not claim that the experience reported here is objective or independent, and the reader must take these comments with this caveat.

First, the author found the ability to talk explicitly about supports in Frame Logic quite intuitive. In particular, the author found it easy to modify pre or postconditions with conjuncts involving supports compared to their experience with thinking about annotations in separation logic. For example, in their first attempt at encoding the merge function (from merge sort), the precondition that they wrote did not specify that the supports of the two lists being merged had to be disjoint and so the program was unable to verify, realizing the cause of this error the condition that the lists be disjoint was simply added as an additional conjunct to the precondition. Similarly, in the partitioning function for quicksort, adding to the post condition that the two resulting lists are disjoint just required the addition of one extra conjunct.

These two examples also highlight the general approach that they found themself taking in the development of the benchmark suite. They first translated the programs from the work on Dryad [Pek et al. 2014] (a Separation Logic variant), and wrote initial specifications mirroring those present in that work. The translation required some rewriting owing to differences in the logics, such as the use of recursively defined functions to compute the minimum or maximum keys stored in a data structure. They then ran the tool and manually investigated the basic blocks which did not verify. This allowed them to identify statements that were causing specific postconditions to fail to verify, and they would then modify the specifications appropriately. Similarly, they also identified insufficient specifications and fixed them. This process crucially used Relaxed Postconditions (and a variant of it with no checks on the support at all, called *supportless* postconditions) as it allowed specifying weaker postconditions whose support may not be equal to the entire set of allocated locations in the post state, and strengthening the postcondition gradually.

This process, and the automated verification, demonstrate the strengths of Frame Logic; allowing the mental separation of dealing with support from other correctness conditions being encoded, and the ability to use the tool to quickly iterate and modify conditions to understand why a piece of code will not verify and then be able to quickly and easily update conditions and re-verify to see if the issue was fixed.

7 RELATED WORK

The work on Frame Logic [Murali et al. 2023, 2020] is the most relevant prior work, which we have discussed at length in the paper. There are many other logics [Banerjee et al. 2008; Bobot and Filliâtre 2012; Hobor and Villard 2013; Kassios 2006; Pek et al. 2014; Reynolds 2002; Smans et al. 2009] for heaps that make different choices in terms of whether heaplets of formulae are implicit, whether formulas have unique heaps, etc. The most popular among these is Separation logic [Demri and Deters 2015; O'Hearn 2012; O'Hearn et al. 2001; Reynolds 2002] which has implicit but non-unique heaplets for formulas. Dynamic Frames [Kassios 2006, 2011] and related approaches like Region Logic [Banerjee and Naumann 2013; Banerjee et al. 2008, 2013] allow users to explicitly specify heaplets and uses them to reason about the program's modifications. The work on Implicit Dynamic Frames [Leino and Müller 2009; Parkinson and Summers 2011; Smans et al. 2009, 2012] combines the ideas of implicit heaplets in separation logic and explicitly accessible heaplets in dynamic frames, resulting in implementations such as Chalice [Parkinson and Summers 2011] and Viper [Müller et al. 2016]. There is also work on translating VeriFast [Jacobs et al. 2011] predicates into Implicit Dynamic Frames [Jost and Summers 2014].

There is rich prior work on reasoning with separation logic. A first category of prior works develop fragments with decidable reasoning [Berdine et al. 2006, 2004, 2005; Cook et al. 2011; Navarro Pérez and Rybalchenko 2011; Pagel 2020; Pérez and Rybalchenko 2013; Piskac et al. 2013]. There is also work on decidability of heap properties specifiable in EPR (Effectively Propositional Reasoning) [Itzhaky et al. 2014a, 2013, 2014b]. A second category of works translate separation logic to first-order logic and use reasoning mechanisms for FOL [Chin et al. 2007; Löding et al. 2018; Madhusudan et al. 2012; Pek et al. 2014; Piskac et al. 2013, 2014a,b; Qiu et al. 2013; Suter et al. 2010]. There are also techniques that reason with separation logic using cyclic proofs [Brotherston et al. 2011; Ta et al. 2016].

The work in [Bobot and Filliâtre 2012] develops a logic similar to FL and uses Why3 [Filliâtre and Paskevich 2013a,b] to reason with verification conditions.

We use in our work the reasoning technique of natural proofs [Löding et al. 2018; Madhusudan et al. 2012; Pek et al. 2014; Qiu et al. 2013]. The technique of quantifier instantiation used by natural proofs is similar to many other unfolding-based techniques in the literature [Jacobs et al. 2011; Kaufmann and Moore 1997; Leino 2012; Nguyen and Chin 2008; Suter et al. 2010].

The work on FL formulates a fragment of separation logic called Precise Separation Logic which has determined, unique heaplets and provides a translation of the fragment to FL. Other prior literature has also studied fragments with determined heaplets [O'Hearn et al. 2004; Pek et al. 2014; Qiu et al. 2013] as it is easier to develop automated reasoning for them. We believe that the techniques developed in this work can open up new pathways to automating separation logics.

8 CONCLUSIONS

Frame logic is based on extending first-order logic with a support operator whose semantics follows a similar philosophy of tight heaplets developed in separation logic. While automation for separation logic has been challenging, especially due to second-order operators such as the magic wand, frame logic was proposed in previous work as a logic that is closed under weakest preconditions without the need for such second-order operators.

This paper realizes the vision of bringing automated FO logic reasoning to frame logic, showing that program verification reasoning against frame logic specifications can in fact be realized using practical heuristics that use SMT solvers and that are furthermore FO-complete.

We also believe that frame logic and its automation can be used as a pathway for other logics to be reasoned with effectively using translations. To this end, we have shown an alternate semantics for a powerful separation logic that can be embedded into frame logic.

In order to reason with first-order logic with recursive definitions, recent work [Murali et al. 2022] has proposed inductive lemma synthesis algorithms that work in tandem with natural proofs. Exploring how to incorporate lemma synthesis into our verification framework could alleviate the current burden on the programmer to write these inductive lemmas (described for our benchmarks in Section 6).

We believe that our alternate semantics for separation logic *SL-FL* is worthy of further study. In particular, since we have shown that its reasoning can be automated using FO reasoning, a track in an SL competition like SLComp [Sighireanu 2021] would be interesting to encourage competitive tools development.

Finally, it would be interesting to see whether frame logic and its automated reasoning explored in this paper can be adapted to reasoning with *incorrectness logics* [O'Hearn 2019], finding ways to use first-order reasoning and SMT reasoning to find errors in programs.

REFERENCES

Alfred V. Aho and Jeffrey D. Ullman. 1979. Universality of Data Retrieval Languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 110–119. https://doi.org/10.1145/567752.567763

Anindya Banerjee and David A. Naumann. 2013. Local Reasoning for Global Invariants, Part II: Dynamic Boundaries. *J. ACM* 60, 3, Article 19 (jun 2013), 73 pages. https://doi.org/10.1145/2485981

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2008. Regional Logic for Local Reasoning about Global Invariants. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 387–411.

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2013. Local Reasoning for Global Invariants, Part I: Region Logic. *J. ACM* 60, 3, Article 18 (June 2013), 56 pages. http://doi.acm.org/10.1145/2485982

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO'05)*. Springer-Verlag, Berlin, Heidelberg, 115–137. https://doi.org/10.1007/11804192_6

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science* (Chennai, India) *(FSTTCS'04)*. Springer-Verlag, Berlin, Heidelberg, 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 52–68. https://doi.org/10.1007/11575467_5

François Bobot and Jean-Christophe Filliâtre. 2012. Separation Predicates: A Taste of Separation Logic in First-Order Logic. In *Formal Methods and Software Engineering*, Toshiaki Aoki and Kenji Taguchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–181.

Rémi Brochenin, Stéphane Demri, and Etienne Lozes. 2008. On the Almighty Wand. In *Computer Science Logic*, Michael Kaminski and Simone Martini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 323–338.

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE'11)*. Springer-Verlag, Berlin, Heidelberg, 131–146. http://dl.acm.org/citation.cfm?id=2032266.2032278

Ashok K. Chandra and David Harel. 1980. Structure and Complexity of Relational Queries. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science (SFCS '80)*. IEEE Computer Society, USA, 333–347. https://doi.org/10.1109/SFCS.1980.41

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2007. Automated Verification of Shape, Size and Bag Properties. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS '07)*. IEEE Computer Society, USA, 307–320. https://doi.org/10.1109/ICECCS.2007.17

Byron Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *Proceedings of the 22nd International Conference on Concurrency Theory* (Aachen, Germany) *(CONCUR'11)*. Springer-Verlag, Berlin, Heidelberg, 235–249.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Stéphane Demri and Morgan Deters. 2015. Separation logics and modalities: a survey. *Journal of Applied Non-Classical Logics* 25 (2015), 50–99.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013a. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Jean-Christophe Filliâtre and Andrei Paskevich. 2013b. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2429069.2429131

Neil Immerman. 1982. Relational Queries Computable in Polynomial Time (Extended Abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (San Francisco, California, USA) *(STOC '82)*. Association for Computing Machinery, New York, NY, USA, 147–152. https://doi.org/10.1145/800070.802187

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014a. Modular Reasoning About Heap Paths via Effectively Propositional Formulas. In *Proceedings of the 41st ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 385–396. https://doi.org/10.1145/2535838.2535854

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning About Reachability in Linked Data Structures. In *Proceedings of the 25th International Conference on Computer Aided Verification* (Saint Petersburg, Russia) *(CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 756–772. https://doi.org/10.1007/978-3-642-39799-8_53

Shachar Itzhaky, Nikolaj Bjørner, Thomas Reps, Mooly Sagiv, and Aditya Thakur. 2014b. Property-Directed Shape Analysis. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'14)*. Springer-Verlag, Berlin, Heidelberg, 35–51. https://doi.org/10.1007/978-3-319-08867-9_3

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 41–55.

Daniel Jost and Alexander J. Summers. 2014. An Automatic Encoding from VeriFast Predicates into Implicit Dynamic Frames. In *Verified Software: Theories, Tools, Experiments*, Ernie Cohen and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 202–221.

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer-Verlag, Berlin, Heidelberg, 268–283.

I. T. Kassios. 2011. The Dynamic Frames Theory. *Form. Asp. Comput.* 23, 3 (May 2011), 267–288. https://doi.org/10.1007/s00165-010-0152-5

Matt Kaufmann and J. S. Moore. 1997. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. Softw. Eng.* 23, 4 (April 1997), 203–213. https://doi.org/10.1109/32.588534

K. Rustan M. Leino. 2012. Automating Induction with an SMT Solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (Philadelphia, PA) *(VMCAI'12)*. Springer-Verlag, Berlin, Heidelberg, 315–331. https://doi.org/10.1007/978-3-642-27940-9_21

K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 378–393. https://doi.org/10.1007/978-3-642-00590-9_27

Leonid Libkin. 2004. *Elements of Finite Model Theory*. Springer. https://doi.org/10.1007/978-3-662-07003-1

Christof Löding, P. Madhusudan, and Lucas Peña. 2018. Foundations for natural proofs and quantifier instantiation. *PACMPL* 2, POPL (2018), 10:1–10:30. https://doi.org/10.1145/3158098

P. Madhusudan, Xiaokang Qiu, and Andrei Ştefănescu. 2012. Recursive Proofs for Inductive Tree Data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 123–136. https://doi.org/10.1145/2103656.2103673

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583* (St. Petersburg, FL, USA) *(VMCAI 2016)*. Springer-Verlag, Berlin, Heidelberg, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Adithya Murali, Lucas Peña, Eion Blanchard, Christof Löding, and P. Madhusudan. 2022. Model-Guided Synthesis of Inductive Lemmas for FOL with Least Fixpoints. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 191 (oct 2022), 30 pages. https://doi.org/10.1145/3563354

Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. 2023. A First-Order Logic with Frames. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 7 (may 2023), 44 pages. https://doi.org/10.1145/3583057

Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. 2020. A First-Order Logic with Frames. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 515–543.

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation Logic + Superposition Calculus = Heap Theorem Prover. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 556–566.

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct 1979), 245–257. https://doi.org/10.1145/357073.357079

Huu Hai Nguyen and Wei-Ngan Chin. 2008. Enhancing Program Verification with Lemmas. In *Proceedings of the 20th International Conference on Computer Aided Verification* (Princeton, NJ, USA) *(CAV '08)*. Springer-Verlag, Berlin, Heidelberg, 355–369. https://doi.org/10.1007/978-3-540-70545-1_34

Peter W. O'Hearn. 2012. A Primer on Separation Logic (and Automatic Program Verification and Analysis). In *Software Safety and Security - Tools for Analysis and Verification*, Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 33. IOS Press, 286–318. https://doi.org/10.3233/978-1-61499-028-4-286

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. https://doi.org/10.1145/3371078

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, London, UK, UK, 1–19. http://dl.acm.org/citation.cfm?id=647851.737404

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2004. Separation and Information Hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. ACM, New York, NY, USA, 268–280. https://doi.org/10.1145/964001.964024

Jens Pagel. 2020. *Decision procedures for separation logic: beyond symbolic heaps.* Ph. D. Dissertation. Wien.

Matthew J. Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 439–458.

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2013. Separation Logic Modulo Theories. In *Programming Languages and Systems (APLAS)*. Springer International Publishing, Cham, 90–106.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification* (Saint Petersburg, Russia) *(CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014a. Automating Separation Logic with Trees and Data. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'14)*. Springer-Verlag, Berlin, Heidelberg, 711–728.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014b. GRASShopper. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–139.

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and P. Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/2491956.2462169

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

Mihaela Sighireanu. 2021. SL-COMP: Competition of Solvers for Separation Logic: Report on the Third Edition. *Int. J. Softw. Tools Technol. Transf.* 23, 6 (dec 2021), 895–903. https://doi.org/10.1007/s10009-021-00628-w

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 148–172. https://doi.org/10.1007/978-3-642-03013-0_8

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 2 (May 2012), 58 pages. https://doi.org/10.1145/2160910.2160911

Philippe Suter, Mirco Dotta, and Viktor Kunćak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1706299.1706325

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 659–676. https://doi.org/10.1007/978-3-319-48989-6_40

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285 – 309.

Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (San Francisco, California, USA) *(STOC '82)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/800070.802186

A OPERATIONAL SEMANTICS

$\bot \xrightarrow{*} \bot$
$(S,H,A) \xrightarrow{x:=y} (S[x \mapsto y],H,A)$
$(S,H,A) \xrightarrow{x:=c} (S[x \mapsto c],H,A)$
$(S,H,A) \xrightarrow{v:=be} (S[v \mapsto be],H,A)$
$(S,H,A) \xrightarrow{x:=y.f} (S[x \mapsto f(y)],H,A)$, if $S(y) \in A$
$(S,H,A) \xrightarrow{x:=y.f} \bot$, if $S(y) \notin A$
$(S,H,A) \xrightarrow{v:=y.d} (S[v \mapsto d(y)],H,A)$, if $S(y) \in A$
$(S,H,A) \xrightarrow{v:=y.d} \bot$, if $S(y) \notin A$
$(S,H,A) \xrightarrow{y.f:=x} (S,H[f \mapsto f[S(y) \mapsto S(x)]])$, if $S(y) \in A$
$(S,H,A) \xrightarrow{y.f:=x} \bot$, if $S(y) \notin A$
$(S,H,A) \xrightarrow{y.f:=c} (S,H[f \mapsto f[S(y) \mapsto S(c)]])$, if $S(y) \in A$
$(S,H,A) \xrightarrow{y.f:=c} \bot$, if $S(y) \notin A$
$(S,H,A) \xrightarrow{y.d:=v} (S,H[f \mapsto d[S(y) \mapsto S(v)]])$, if $S(y) \in A$
$(S,H,A) \xrightarrow{y.d:=v} \bot$, if $S(y) \notin A$
$(S,H,A) \xrightarrow{y.d:=be} (S,H[f \mapsto d[S(y) \mapsto S(be)]])$, if $S(y) \in A$
$(S,H,A) \xrightarrow{y.d:=be} \bot$, if $S(y) \notin A$
$(S,H,A) \xrightarrow{alloc(x)} (S[x \mapsto a],H[f \mapsto f[a \mapsto default_f], d \mapsto d[a \mapsto default_d]], A \cup \{a\})$ for every pointer f
A
$(S[x \mapsto a],H,A) \xrightarrow{free(x)} (S,H,A \backslash \{a\})$, if $a \in A$
$(S[x \mapsto a],H,A) \xrightarrow{free(x)} \bot$, if $a \notin A$
$(S,H,A) \xrightarrow{assume(\eta)} (S,H,A)$ if $S \models \eta$
$(S,H,A) \xrightarrow{\text{if } \eta \text{ then } P \text{ else } Q} (S',H',A')$ if $(S,H) \models \eta$ and $(S,H,A) \xrightarrow{P} (S',H',A')$
$(S,H,A) \xrightarrow{\text{if } \eta \text{ then } P \text{ else } Q} (S',H',A')$ if $(S,H) \not\models \eta$ and $(S,H,A) \xrightarrow{Q} (S',H',A')$
$(S,H,A) \xrightarrow{P;Q} (S'',H'',A'')$ if $(S,H,A) \xrightarrow{P} (S',H',A')$ and $(S',H',A') \xrightarrow{Q} (S'',H'',A'')$
$(S,H,A) \xrightarrow{\bar{b}:=g(\bar{a})} (S',H',A')$ if $(S,H,A) \rightarrow (S_g,H,A)$ and $(S_g,H,A) \xrightarrow{g} (S'_{call},H',A')$ and $(S'_{call},H',A') \rightarrow (S',H',A')$. Let g be defined $\bar{q} := g(\bar{p})$. S_g has new variables $\bar{p} \mapsto \bar{a}$. g is then inlined. S' is S'_{call} without the $\bar{p}$ variables. This is a call-by-value method of passing arguments to g.

B TRANSLATION OF FL TO FORD

Translation of FL to FORD is given in Figure 10.

C DETAILS FOR SECTION 5: BASE LOGIC

C.1 Proof of Lemma 5.1

Proof. (1) By structural induction on α. The only rules that create new elements are those for $x \xrightarrow{f} y$ and $\exists$ formulas. In both cases, the newly added elements are checked to be be elements of $dom(h)$.
(2) By structural induction on α. In definition of support for $x \xrightarrow{y}$, note that if support is $\{s(x)\}$, then since h and h' agree on $s(x)$, $s(x)$ is in $dom(h')$ as well, and hence $Supp(\alpha, s, h') = s(x)$.

$$
\begin{aligned}
\Pi(\top) &= \top \\
\Pi(\bot) &= \bot \\
\Pi(c) &= c \\
\Pi(x) &= x \\
\Pi(f(t_1,\ldots,t_n)) &= f(\Pi(t_1),\ldots,\Pi(t_n)) \text{ for any function } f \\
\Pi(t_1 = t_2) &= (\Pi(t_1) = \Pi(t_2)) \\
\Pi(R(t_1,\ldots,t_n)) &= R(\Pi(t_1),\ldots,\Pi(t_n)) \text{ for any relation } R \\
\Pi(\alpha \wedge \beta) &= \Pi(\alpha) \wedge \Pi(\beta) \\
\Pi(\alpha \vee \beta) &= \Pi(\alpha) \vee \Pi(\beta) \\
\Pi(\neg\varphi) &= \neg\Pi(\varphi) \\
\Pi(ite(\gamma : \alpha, \beta)) &= ite(\Pi(\gamma) : \Pi(\alpha), \Pi(\beta)) \\
\Pi(ite(\gamma : t_1, t_2)]) &= ite(\Pi(\gamma) : \Pi(t_1), \Pi(t_2)) \\
\Pi(Sp(\top)) = \Pi(Sp(\bot)) &= \phi \\
\Pi(Sp(c)) = \Pi(Sp(x)) &= \phi \\
\Pi(Sp(f(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n}\{t_i\} \cup \bigcup_{i=1}^{n} \Pi(Sp(t_i)) \text{ if } f \in \mathcal{F}_m \\
\Pi(Sp(f(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n} \Pi(Sp(t_i)) \text{ if } f \notin \mathcal{F}_m \\
\Pi(Sp(Sp(\varphi))) &= \Pi(Sp(\varphi)) \\
\Pi(Sp(Sp(t))) &= \Pi(Sp(t)) \\
\Pi(Sp(t_1 = t_2)) &= \Pi(Sp(t_1)) \cup \Pi(Sp(t_2)) \\
\Pi(Sp(R(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n} \Pi(Sp(t_i)) \text{ for } R \in \mathcal{R} \\
\Pi(Sp(\alpha \wedge \beta)) &= \Pi(Sp(\alpha)) \cup \Pi(Sp(\beta)) \\
\Pi(Sp(\alpha \vee \beta)) &= \Pi(Sp(\alpha)) \cup \Pi(Sp(\beta)) \\
\Pi(Sp(\neg\varphi)) &= \Pi(Sp(\varphi)) \\
\Pi(Sp(ite(\gamma : \alpha, \beta))) &= \ \textit{if}\, \gamma \textit{ then } \Pi(Sp(\gamma)) \cup \Pi(Sp(\alpha)) \textit{ else } \Pi(Sp(\gamma)) \cup \Pi(Sp(\beta)) \\
\Pi(Sp(ite(\gamma : t_1, t_2))) &= \ \textit{if}\, \gamma \textit{ then } \Pi(Sp(\gamma)) \cup \Pi(Sp(t_1)) \textit{ else } \Pi(Sp(\gamma)) \cup \Pi(Sp(t_2)) \\
\Pi(Sp([\alpha])) &= \phi \\
\Pi(Sp(I(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n} \Pi(Sp(t_i)) \cup SpI(t_1,\ldots,t_n) \text{ for } I \in \mathcal{I}, \\
&\quad \text{where } SpI(\bar{x}) :=_{lfp} \Pi(Sp(\rho_I(\bar{x}))) \text{ and } I(\bar{x}) :=_{lfp} \rho_I(\bar{x}) \\
\Pi([\alpha]) &= \Pi(\alpha)
\end{aligned}
$$

Fig. 10. Translation from FL to FORD. For an FL formula α, the translated FORD formula is $\Pi(\alpha)$, recursively defined above.

In definition of support for $\exists y.(x \xrightarrow{f} y : \alpha)$, notice that $S = s(x)USupp(alpha, s', h)$, where $s' = s[y \mapsto h(f)(s(x))]$. But since $s(x)$ is in S, h' agrees with h on it, and hence $h'(f)(s(x)) = h(f)(s(x))$. Hence $Supp(\alpha, s, h') = \{s(x)\} \cup Supp(\alpha, s', h) = Supp(\alpha, s, h)$ (by induction hypothesis). The rest of the cases are trivial.

(3) Follows from (2) since $h \downharpoonright S$ agrees with h on S.
(4) Heaplet h agrees with h_1 on $dom(h_1)$, and so $Supp(\alpha, s, h) = dom(h_1)$. Heaplet h agrees with h_2 on $dom(h_2)$, and so $Supp(\alpha, s, h) = dom(h_2)$. Hence $dom(h_1) = dom(h_2)$.

□

C.2 Proof of Lemma 5.2

(1) Easily proved by structural induction on α.
(2) Assume there are two subheaplets that satisfy α. By (1), the supports of α in each of them would be their own domains. But by Lemma 5.1 (4), the heaplets of these domains must be the same. Hence the heaplets are identical. □

D DETAILS FOR SECTION 5: EXTENDED LOGIC *SL-FL*

The semantics of this logic extends the semantics on the base logic in the following ways. First, we assume that each background sort is a complete lattice (for flat sorts like arithmetic, we can introduce a bottom element and a top element to obtain a complete lattice). Next, we treat the equations defining *Supp* in Figure 6 as recursive definitions with least fixpoint semantics, with $\subseteq$ over sets as ordering. We modify $\mathit{Supp}(\mathit{ite}(\gamma, \alpha, \beta), s, h)$ to be $\mathit{Supp}(\gamma) \cup \mathit{Supp}(\alpha, s, h)$ if $s, h' \models \gamma$ for some subheap h' of h, and to be $\mathit{Supp}(\gamma) \cup \mathit{Supp}(\beta)$ otherwise. We add support definitions for recursively defined predicates and functions:

$$\mathit{Supp}(R(\overline{x}), s, h) = \mathit{Supp}(\rho_R(\overline{x}), s, h)$$

$$\mathit{Supp}(F(\overline{x}), s, h) = \mathit{Supp}(\mu_F(\overline{x}), s, h)$$

where $R(\overline{x}) = \rho_R(\overline{x})$ and $F(\overline{x}) = \mu_F(\overline{x})$ are the definitions of R and F, respectively. We can show that the least fixpoint of the above equations defining the map *Supp* satisfies the properties described in Lemma 5.1 for the extended logic as well.

The semantics of *SL-FL*is now easy to define. We modify the semantics given in Figure 7 as follows, redefining the semantics for *ite* and defining the semantics for recursively defined predicates:

$$\begin{array}{rll}
(s,h) \models \mathit{ite}(\alpha', \alpha, \beta) & \mathit{iff} & \text{there exist } H_1, H_2 : H_1 \cup H_2 = \mathit{dom}(h), \mathit{Supp}(s, h \downharpoonright H_1) = H_1, \text{ and} \\
& & ((s, h \downharpoonright H_1) \models \alpha' \text{ and } (s, h \downharpoonright H_2) \models \alpha) \text{ or } ((s, h \downharpoonright H_1) \not\models \alpha' \text{ and } (s, h \downharpoonright H_2) \models \beta) \\
(s,h) \models R(\overline{x}) & \mathit{iff} & (s,h) \models \rho(\overline{x}), \ \text{ where } R\text{'s definition is } R(\overline{x}) = \rho(\overline{x})
\end{array}$$

We can prove lemmas analogous to Lemma 5.2 for the extended logic.

Finally, we can translate the extended logic to frame logic as well, modifying the translation given in Figure 8 for *ite* formulas and translating definitions:

$$\begin{array}{rcl}
\Pi(\mathit{ite}(\alpha', \alpha, \beta)) & = & \mathit{ite}(\Pi(\alpha'), \Pi(\alpha), \Pi(\beta)) \\
\Pi(p(\overline{t})) & = & p(\overline{t}) \\
\Pi(R(\vec{x})) & = & R(\vec{x}) \\
\Pi(x.f)) & = & f(x) \\
\Pi(f(\overline{t})) & = & f(\Pi(\overline{t})) \\
\Pi(\mathit{ite}(\alpha, t, t')) & = & \mathit{ite}(\alpha, \Pi(t), \Pi(t') \\
\Pi(F(\overline{t})) & = & F(\Pi(\overline{t})) \\
\Pi(\ R(\vec{x}) :=_{\mathit{lfp}} \rho(\vec{x})\ ) & = & R(\vec{x}) :=_{\mathit{lfp}} \Pi(\rho(\vec{x})) \\
\Pi(\ F(\vec{x}) :=_{\mathit{lfp}} \mu(\vec{x})\ ) & = & R(\vec{x}) :=_{\mathit{lfp}} \Pi(\mu(\vec{x}))
\end{array}$$

We can now prove the lemma analogous to Lemma 5.3:

Lemma D.1. *Let g be a global heap (a heaplet where with domain Loc). Let α be an SL-FLformula.*

- $g \models \Pi(\alpha)$ *iff there exists a heaplet h of g such that $(s, h) \models \alpha$.*
- *If $g \models \Pi(\alpha)$, then $\mathit{Supp}(\alpha, s, g) = \mathit{Sp}(\Pi(\alpha))$.* □

E DETAILS OF EVALUATION

E.1 Definitions of Datastructures

The definitions of the data structures used in the benchmarks and use the explicit support and cloud operators. The definition for singly-linked lists is as given below:

$$\mathit{List}(x) := \mathit{ite}(x = \mathit{nil}, \top, \mathit{List}(\mathit{next}(x)) \wedge x \notin \mathit{Sp}(\mathit{List}([\mathit{next}(x)]))$$

and the definition for doubly-linked lists is similar, with the addition of a constraint that if $next(x)$ is not nil then its previous pointer should point back to x,

$$\begin{aligned}Dll(x) := ite(&x = nil, \top, ite(next(x) = nil, \top,\\ &x = prev(next(x)) \land Dll(next(x)) \land x \notin Sp(Dll([next(x)]))))\end{aligned}$$

Next, to define sorted lists, we rely on an auxiliary definition of the minimum element in a list and then a (non-decreasing) sorted list is simply a list where the front key is no greater than the minimum value of its tail, which is itself also a sorted list:

$$\begin{aligned}Min(x) &:= ite(x = nil, +\infty, ite(key(x) < Min(next(x)), key(x), Min(next(x))))\\ Sorted(x) &:= ite(x = nil, \top,\\ &\qquad Sorted(next(x)) \land x \notin Sp(Sorted([next(x)])) \land key(x) \le Min(next(x)))\end{aligned}$$

In our implementation of this definition, $+\infty$ is simply an additional variable that is treated as having an infinitely large value.

For circular lists, we define list segments using the $Lseg(\cdot)$ definition and we then define a circular list as empty or reaching itself through its next pointer,

$$\begin{aligned}Lseg(x, y) &:= ite(x = y, \top, ite(x = nil, \bot,\\ &\qquad Lseg(next(x), y) \land x \notin Sp(Lseg([next(x)], y))))\\ Circ(x) &:= ite(x = nil, \top, Lseg(next(x), x) \land x \notin Sp(Lseg([next(x)], x)))\end{aligned}$$

Our definition for binary search trees, like for sorted lists, relies on a definition of the minimum and maximum values in a tree and the binary search property is specified by requiring the maximum element in the left subtree to be less than the key of the root and the minimum element in the right subtree to be greater than the key of the root. The definitions of *Min* and *Max* are similar to that for sorted lists and so are omitted; we note that the default value for *Max* if x is *nil* is $-\infty$. Then, the definition of a binary search tree is

$$\begin{aligned}BST(x) := ite(&x = nil, \top,\\ &BST(left(x)) \land Max(left(x)) < key(x) \land BST(right(x))\\ &\land key(x) < Min(right(x)) \land x \notin Sp(BST([left(x)]))\\ &\land x \notin Sp(BST([right(x)])) \land Sp(BST([left(x)])) \cap Sp(BST([right(x)])) = \emptyset)\end{aligned}$$

In addition to our normal requirement that x not be in the support of either of the subtrees we also specify here that the supports of the subtrees must be disjoint; together these properties express that x is indeed a tree.

The definitions for Treaps and Red-Black Trees are similar to this definition for binary search trees, augmented with their appropriate additional constraints, the max heap property on the priorities in the Treap and the red-black property for Red-Black trees. The implementation of the max heap property for the Treap is in the same fashion as the binary search property for binary-search trees, except that we just require that the priority of the root be greater than the maximum priority of either subtree. For red-black trees we add a recursive definition which determines the (maximum) black-height of a tree and then a tree is a red-black tree if both subtrees have the same black height and the root is black or both children are black.

E.1.1 Identifying Buggy Programs. To evaluate the detection of buggy programs using the FLV tool, we took five of our benchmarks and removed a conjunct from the pre-condition. Since during the construction of our benchmark suite, one common issue was omitting necessary constraints on the support (such as that support for the lists in singly-linked list append be disjoint), we have included several buggy programs that omit these support conditions, as well as other programs omitting non-support conditions. In particular, our buggy benchmarks are derived from the Singly-Linked List append benchmark, where we removed the pre-condition that the two lists be disjoint; the Sorted Merge function (used in Merge Sort), where we similarly removed the condition that the two lists be disjoint; the Sorted Concat function (used in Quicksort), where we removed the condition that the maximum element of the first list be less than the minimum element of the second; the Circular List Find End function (used in Circular List Delete), where we removed a pre-condition that the head and tail be distinct; and the BST Remove Root function (used in BST Delete), where we removed the condition that the tree be non-nil. All of these bugs were either bugs we encountered in the process of building our benchmark suite or are similar to bugs we encountered in the process.

With the bug in the singly-linked list append, the tool finished and reported that the program was not verified within about four minutes. Specifically, as expected, the block representing the base case of the recursion succeed in verification and the block containing the recursive call failed. With the bug in the sorted merge program, the two base cases still verify but attempts at verifying either of the blocks containing the recursive cases, timeout at twenty minutes; in fact within the time limit it does not report either success or failure in verifying the pre-condition of the recursive call. The bug in the sorted concatenation similarly verifies its base case and times out attempting to verify the recursive case, though in this case, the pre-condition of the recursive call is satisfied. For the bug in circular list find end, the base case in this case does not satisfy the post-condition and the tool reports that this block does not verify within ten seconds and the tool times out attempting to verify the recursive case. With this bug, the tool is able, within the time limit, to report that it could not verify the pre-condition of the recursive call. Finally, for the BST remove root bug, this function has three base cases, one of which is valid and the other two of which are not and the tool reports that they do not verify within about thirty seconds each; again the tool times out attempting to verify the recursive case and does report success of failure of verifying the pre-condition.