We propose a novel logic, Frame Logic (FL), that extends first-order logic and recursive definitions with a construct $Sp(\cdot)$ that captures the implicit supports of formulas—the precise subset of the universe upon which their meaning depends. Using such supports, we formulate proof rules that facilitate frame reasoning elegantly when the underlying model undergoes change. We show that the logic is expressive by capturing several data-structures and also exhibit a translation from a precise fragment of separation logic to frame logic. Finally, we design a program logic based on frame logic for reasoning with programs that dynamically update heaps that facilitates local specifications and frame reasoning. This program logic consists of both localized proof rules as well as rules that derive the weakest tightest preconditions in frame logic.

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

Additional Key Words and Phrases: Frame reasoning, program verification, program logics, heap verification, first-order logic, first-order logic with recursive definitions

ACM Reference format:
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

1 INTRODUCTION

Program logics for expressing and reasoning with programs that dynamically manipulate heaps is an active area of research. The research on separation logic has argued convincingly that it is highly desirable to have localized logics that talk about small states (heaplets rather than the global heap) and the ability to do frame reasoning. Separation logic achieves this objective by having a tight heaplet semantics and using special operators, primarily a separating conjunction operator $*$ and a separating implication operator (the magic wand $\rightarrow$).
In this article, we ask a fundamental question: Can logics such as FOL and FOL with recursive definitions be extended to support localized specifications and frame reasoning? Can we utilize such logics for reasoning effectively with programs that dynamically manipulate heaps with the aid of local specifications and frame reasoning?

The primary contribution of this article is to endow the first-order logic involving recursive definitions (with least fixpoint semantics) with frames and frame reasoning.

A formula in **first-order logic with recursive definitions (FO-RD)** can be naturally associated with a **support**—the subset of the universe that determines its truth. By using careful syntax such as guarded quantification (which continue to have a classical interpretation), we can in fact write specifications in FO-RD that have very precise supports. For example, we can write the property that $x$ points to a linked list using a formula $\text{list}(x)$ written purely in FO-RD so its support is precisely the locations constituting the linked list.

In this article, we define an extension of FO-RD called **Frame Logic (FL)**, where we allow a new operator $\text{Sp}(\alpha)$. For an FO-RD formula $\alpha$, $\text{Sp}(\alpha)$ evaluates to the support of $\alpha$. Logical formulas thus have access to supports and can use this to separate supports and do frame reasoning. For instance, the logic can now express that two lists are disjoint by asserting that $\text{Sp}(\text{list}(x)) \cap \text{Sp}(\text{list}(y)) = \emptyset$. It can then reason that in such a program heap configuration, if the program manipulates only the locations in $\text{Sp}(\text{list}(y))$, then $\text{list}(x)$ would continue to be true, using simple frame reasoning.

Specifications can be written succinctly using macros such as $\text{Star}(\alpha, \beta) = \alpha \land \beta \land \text{Sp}(\alpha) \cap \text{Sp}(\beta) = \emptyset$. When disjointness is not known, FL allows us to simply write the formula $\text{list}(x) \land \text{list}(y)$ to represent lists that *may or may not* be overlapping. Such formulas are very difficult to write in Separation Logic.

The addition of the support operator to FO-RD yields a very natural logic for expressing specifications. First, formulas in FO-RD have the same meaning when viewed as FL formulas. For example, $f(x) = y$ (written in FO-RD as well as in FL) is true in any model that has $x$ mapped by $f$ to $y$, instead of a specialized “tight heaplet semantics” that demands that $f$ be a partial function with the domain only consisting of the location $x$. The fact that the support of this formula contains only the location $x$ is important, of course, but is made accessible using the support operator, i.e., $\text{Sp}(f(x) = y)$ gives the singleton set containing the element interpreted for $x$.\(^1\) Second, properties of supports can be naturally expressed using set operations. To state that the lists pointed to by $x$ and $y$ are disjoint, we do not need special operators (such as the $\ast$ operator in separation logic) and can express it as $\text{Sp}(\text{list}(x)) \cap \text{Sp}(\text{list}(y)) = \emptyset$. Third, when used to annotate programs, pre/post specifications for programs written in FL can be made *implicitly* local by interpreting their supports to be the localized heaplets accessed and modified by programs, yielding frame reasoning akin to program logics that use separation logic. Finally, as we show in this article, the weakest precondition of specifications across basic loop-free paths can be expressed in FL, making it an expressive logic for reasoning with programs. Separation logic, however, introduces the magic wand operator $\ast \ast$ (which is inherently higher-order) to add enough expressiveness to be closed under weakest preconditions [63].

We define frame logic (FL) as an extension of FO with recursive definitions (FO-RD) that operates over a multi-sorted universe, with a particular foreground sort (used to model locations on the heap on which pointers can mutate) and several background sorts that are defined using separate theories. Supports for formulas are defined with respect to the foreground sort only. A special background sort of *sets* of elements of the foreground sort is assumed and is used to model the supports for formulas. For any FL formula $\varphi$, $\text{Sp}(\varphi)$ captures its *support*, which is a set of elements of the foreground sort. The support intuitively corresponds to the precise subdomain

\[^1\]For a pointer $f$, we denote by $f(x) = y$ that $x$ points to $y$ on $f$, i.e., $x \mapsto y$, whose “footprint” is $\{x\}$ in many heap logics.
of mutable functions (modeling pointers) on which the truth value of $\varphi$ depends. We then prove a frame theorem (Theorem 3.7) that says that changing a model $M$ by changing the interpretation of functions outside the support of $\varphi$ will not affect the truth of $\varphi$. This theorem directly supports frame reasoning; if a model satisfying $\varphi$ is changed such that the changes made are disjoint from the support of $\varphi$, then $\varphi$ will continue to hold. We also show that FL formulas can be translated to vanilla FO-RD logic (without support operators); in other words, the semantics for the support of a formula can be captured in FO-RD itself. Consequently, we can use any FO-RD reasoning mechanism (proof systems [37, 38] or heuristic algorithms such as the natural proof techniques [46, 57, 62, 67]) to reason with FL formulas.

We illustrate our logic using several examples drawn from program verification. We show how to express various data-structure definitions and various measures on them using FL formulas, e.g., linked lists, sorted lists, list segments, binary search trees, AVL trees, lengths of lists, heights of trees, set of keys stored in the data-structure, and so on.

While the sensibilities of our logic are definitely inspired by separation logic, there are some fundamental differences beyond the fact that our logic extends the syntax and semantics of FO-RD with a special support operator and avoids operators such as $\ast$ and $\ast \ast$. In separation logic, there can be many supports of a formula (also called heaplets)—a heaplet for a formula is one that supports its truth. For example, a formula of the form $\alpha \lor \beta$ can have a heaplet that supports the truth of $\alpha$ or one that supports the truth of $\beta$. However, the philosophy that we follow in our design is to have a single support that supports the truth value of a formula, whether it be true or false. Consequently, the support of the formula $\alpha \lor \beta$ is the union of the supports of the formulas $\alpha$ and $\beta$.

The above design choice of the support being determined by the formula has several consequences that lead to a deviation from separation logic. For instance, the support of the negation of a formula $\varphi$ is the same as the support of $\varphi$. The support of the formula $f(x) = y$ and its negation are the same, namely, the singleton set containing the location interpreted for $x$. In separation logic, the heaplet of the corresponding formula is the same as the support in FL, but its negation will include all other heaplets. The choice of having determined supports or heaplets is not new, and there have been several variants and sublogics of separation logics that have been explored. For example, the logic Dryad [57, 62] is a separation logic that insists on determined heaplets to support automated reasoning, and the precise fragment of separation logic studied in the literature [53] defines a sublogic that has (essentially) determined heaplets. The second main contribution in this article is to show that such a precise fragment of separation logic can be translated to frame logic, such that the unique heaplet that satisfies a precise separation logic formula is its support of the corresponding formula in frame logic.

The third main contribution of this article is a program logic based on frame logic for a simple while-programming language that can destructively update heaps. We present two kinds of proof rules for reasoning with such programs annotated with pre- and post-conditions written in frame logic. The first set of rules are local rules that axiomatically define the semantics of the program using the smallest supports for each command. We also develop a frame rule that allows arguing preservation of properties whose supports are disjoint from the heaplet modified by a program. These rules are similar to analogous rules in separation logic. The second set of rules allows us to infer the weakest tightest precondition for a postcondition with respect to non-recursive programs. In separation logic, the corresponding rules for weakest preconditions are often expressed using separating implication (the magic wand operator). Given a small change made to the heap and a postcondition $\beta$, the formula $\alpha \leftarrow \ast \beta$ captures all heaplets $H$ such that if a heaplet that is disjoint from $H$ satisfies $\alpha$, then its union with $H$ satisfies $\beta$. When $\alpha$ describes the change effected by

2The blowup of translation from FL to FO-RD is only linear; this can be seen from Figure 2.
the program, $\alpha \ast \beta$ essentially captures the weakest precondition. However, the magic wand is a very powerful operator that calls for quantifications over heaplets and submodels, and hence involves second-order quantification [14]. In our logic, we show that we can capture the weakest precondition with only first-order quantification, and hence first-order frame logic is closed under weakest preconditions across non-recursive program blocks. This means that when inductive loop invariants are written in FL, reasoning about programs reduces to reasoning in FL. Finally, by translating FL to pure FO-RD formulas, we can use reasoning techniques for FO-RD to reason with FL, and hence programs.

Our work is in large part inspired by our previous work on reasoning with Dryad [45, 57, 62] using FO-reasoning engines, in particular, SMT solvers. Dryad is a separation logic that has, essentially, unique heaplets, and this is crucial in translation to FO without quantifiers over sets (for example, quantifying over two heaplets to reason with $\alpha \ast \beta$). Dryad does not have the magic wand and is therefore not closed under weakest preconditions. We imbibe this philosophy fundamentally in the design of Frame Logic—the support for formulas is uniquely defined. When converting Dryad formulas to FO-RD, the translation has separate components for capturing truth and heaplets (especially heaplets for inductively defined datastructures). Moreover, the correctness of the heaplet definition is not derived or verified to be correct in the works in References [45, 57, 62]. Frame Logic addresses this disadvantage by ensuring that the supports of formulas are systematically captured using first-order logic (by the Frame Logic to FO-RD translation). Reasoning with Dryad proceeds using a verification condition technique based on strongest post, which avoids introducing unnecessary quantification. From our experience in working with both Dryad and Frame Logic, we believe that logical engines to reason with FO-RD formulas derived from Frame Logic will be as successfully amenable to automation as Dryad is, especially when adapted to verification conditions derived using the strongest-post. However, such an extension is considerable effort and we leave it to future work. We provide further discussion in Section 6.2.

In summary, this article presents Frame Logic, an alternative to separation logic that is designed on principles similar to separation logic. The main contributions of this article are:

- A logic, called frame logic (FL), that extends FO-RD with a support operator and supports frame reasoning. We illustrate FL with specifications of various data-structures. We show a translation to equivalent formulas in FO-RD.
- A program logic and proof system based on FL including local rules and rules for computing the weakest tightest precondition. FL reasoning required for proving programs is hence reducible to reasoning with FO-RD.
- A separation logic fragment that can generate only precise formulas and a translation from this logic to equivalent FL formulas.

Article Structure. The article is organized as follows: Section 2 sets up first-order logics with recursive definitions (FO-RD), with a special uninterpreted foreground sort of locations and several background sorts/theories. Section 3 introduces FL, its syntax, its semantics, which includes a discussion of design choices for supports, proves the frame theorem for FL, shows a reduction of FL to FO-RD, and illustrates the logic by defining several data-structures and their properties using FL. Section 4 develops a program logic based on FL, illustrating them with proofs of verification of programs. Section 5 introduces a precise fragment of separation logic and shows its translation to FL. Section 6 discusses comparisons of FL to separation logic and some existing first-order techniques that can be used to reason with FL. Section 7 compares our work with the research literature, and Section 8 has concluding remarks.

This article is an extension of our work at ESOP 2020 [49]. In this work, we present a cleaner operational semantics, a simpler program logic and give detailed proofs of all our results. We also
present detailed comparisons in the Discussion section (Section 6) and the section on Related Work (Section 7).

2 BACKGROUND: FIRST-ORDER LOGIC WITH RECURSIVE DEFINITIONS AND UNINTERPRETED COMBINATIONS OF THEORIES

The base logic upon which we build frame logic is a first-order logic with recursive definitions (FO-RD), where we allow a foreground sort and several background sorts, each with their individual theories (such as arithmetic, sets, arrays, etc.). The logic FO-RD is essentially the same as the classical first-order logic with least fixpoints (FO-lfp) studied in finite model theory and databases from the 1980s [1, 18, 29, 44, 70]. The only difference is that we give names to recursive definitions that have least fixpoint semantics. Logics with inductive definitions over non-monotonic bodies are also studied in the literature, but we do not study such definitions in our article; see Section 7 on related work.

The foreground sort and functions involving the foreground sort are uninterpreted (not constrained by theories). This, hence, can be seen as an uninterpreted combination of theories over disjoint domains. This logic has been defined and used to model heap verification in prior work [45].

We build frame logic over such a framework where supports are modeled as subsets of elements of the foreground sort. When modeling heaps in program verification using logic, the foreground sort is used to model locations of the heap, uninterpreted functions from the foreground sort to foreground sort are used to model pointers, and uninterpreted functions from the foreground sort to the background sort model data fields. Consequently, supports will be subsets of heap locations, which is appropriate, as these are the domains of pointers that change when a program updates a heap.

Formally, we work with a signature $\Sigma = (S; C; F; R; I)$, where $S$ is a finite nonempty set of sorts. $C$ is a set of constant symbols, where each $c \in C$ has some sort $\tau \in S$. $F$ is a set of function symbols, where each function $f \in F$ has a type of the form $\tau_1 \times \cdots \times \tau_m \to \tau$ for some $m$, with $\tau_i, \tau \in S$. The sets $R$ and $I$ are (disjoint) sets of relation symbols, where each relation $R \in R \cup I$ has a type of the form $\tau_1 \times \cdots \times \tau_m$. The set $R$ contains relation symbols whose interpretations are given by a model, while those $I$ are inductively defined using formulas (see below).

We assume that the set of sorts contains a designated “foreground sort” denoted by $\sigma_f$. All the other sorts in $S$ are called background sorts, and for each such background sort $\tau$, we allow the constant symbols, function symbols, and relation symbols to be constrained using an arbitrary first-order theory $T_\tau$, e.g., linear arithmetic.

A formula in FO-RD over such a signature is of the form $(D, \alpha)$, where $D$ is a set of recursive definitions and $\alpha$ is a first-order formula over the given signature. A recursive definition for $R \in I$ is of the form $R(\bar{x}) := \rho_R(\bar{x})$, where $\rho_R(\bar{x})$ is a first-order formula in which relation symbols from $I$ occur only positively, i.e., occur under an even number of negations. We assume $D$ has exactly one definition for every relation symbol in $I$.

The semantics of a formula is standard; the semantics of inductively defined relations are defined to be the least fixpoint that satisfies the relational equations, and the semantics of $\alpha$ is the standard one defined using these semantics for relations. We defer presenting the formal semantics of FO-RD here and present in the next section the formal semantics of Frame Logic, which is an extension of FO-RD.

3 FRAME LOGIC

We now define Frame Logic (FL), the central contribution of this article.

We consider a universe with a foreground sort and various background sorts restricted by individual theories, as described in Section 2. We consider the elements of the foreground sort to be locations and consider supports as sets of locations, i.e., sets of elements of the foreground sort.
We hence introduce a background sort \( \sigma_{S(f)} \) to model sets of elements of sort \( \sigma_f \). We add to \( R \) the relation \( \in : \sigma_f \times \sigma_{S(f)} \) that is interpreted as the usual element relation. We also add standard operations \( \cup, \cap, \sim \) (complement) and the constant \( \emptyset \) with the usual interpretation. We assume that these are constrained by a background theory of sets \( B_{\sigma_{S(f)}} \). We further assume that the signature does not contain any other function or relation symbols involving the sort \( \sigma_{S(f)} \).

For reasoning about changes of the structure over the locations, we assume that there is a nonempty subset \( F_m \subseteq F \) of mutable functions. These functions can be used to model mutable pointer fields in the heap that can be manipulated by a program and thus change. We also require that each \( f \in F_m \) has at least one argument of sort \( \sigma_f \).

Finally, we denote by \( \text{Var}_\tau \) the set of variables of sort \( \tau \) (for \( \tau \in S \)). We denote tuples of variables by \( \overline{x} \).

Frame Logic over uninterpreted combinations of theories is a variant of first-order logic with recursive definitions that has an additional operator \( \text{Sp}(\phi) \) that assigns to each formula \( \phi \) a set of elements (its support or “heaplet” in the context of heaps) in the foreground universe. \( \text{Sp}(\phi) \) is a set of elements.

The intended semantics of \( \text{Sp}(\phi) \) is defined formally as a least fixpoint of a set of equations. This semantics is presented in Section 3.3. In the following, we first define the syntax of the logic and discuss informally the various design decisions for the semantics of supports before proceeding to a formal definition of the semantics.

### 3.1 Syntax of Frame Logic (FL)

The syntax of our logic is given in the grammar in Figure 1. This extends FO-RD with the rule for building support expressions

\[
\text{FL} \quad \varphi ::= \bot \mid T \mid t_\tau = t_\tau \mid R(t_{r_1}, \ldots, t_{r_m}) \mid \varphi \land \varphi \mid \neg \varphi \mid \text{ite}(\gamma : \varphi, \varphi) \mid \exists y : y. \varphi
\]

\( r \in S, R \in \mathcal{R} \cup \mathcal{I} \) of type \( r_1 \times \cdots \times r_m \)

\[
\text{Guards:} \quad \gamma ::= t_\tau = t_\tau \mid R(t_{r_1}, \ldots, t_{r_m}) \mid \gamma \land \gamma \mid \neg \gamma \mid \text{ite}(\gamma : y, \gamma) \mid \exists y : y. \gamma
\]

\( r \in S \setminus \{\sigma_{S(f)}\}, R \in \mathcal{R} \) of type \( r_1 \times \cdots \times r_m \)

\[
\text{Terms:} \quad t_\tau ::= c \mid x \mid f(t_{r_1}, \ldots, t_{r_m}) \mid \text{ite}(y : t_r, t_r) \mid \text{Sp}(\phi) \quad (\text{if} \ r = \sigma_{S(f)}) \mid \text{Sp}(t_\tau) \quad (\text{if} \ r = \sigma_{S(f)})
\]

\[
\text{Recursive definitions:} \quad R(\overline{x}) := \rho_R(\overline{x}) \quad \text{with} \quad R \in \mathcal{I} \quad \text{of type} \quad r_1 \times \cdots \times r_m \quad \text{with} \quad r_i \in S \setminus \{\sigma_{S(f)}\}, \text{FL formula} \rho_R(\overline{x}) \quad \text{where all relation symbols} \quad R' \in \mathcal{I} \quad \text{occur only positively or inside a support expression.}
\]
general, has a different support. Universal formulas $\forall x : y . \alpha$ are shorthands for $\neg(\exists x : y . \neg\alpha)$ as expected, and their supports are defined accordingly.

To illustrate the syntactic constraints, consider, for example, the formula $\exists y : y = \text{next}(x) \land \text{list}(y)$. It is syntactically valid as the guard only references variables and the mutable function $\text{next}$. Since $y = \text{next}(x)$, we can conclude that $x$ points to a list segment ending at $y$, which we denote $\text{lseg}(x, y)$. However, the formula $\exists y : \text{lseg}(x, y) \land \text{list}(y)$ is not syntactically valid in Frame Logic, as the guard contains the recursively defined relation $\text{lseg}$.

Finally, we require that recursively defined relations (we use “inductive definitions” and “recursive definitions” interchangeably) $R \in I$ do not have arguments of sort $\sigma_S(\cdot)$. This is another restriction to ensure the existence of a least fixpoint model in the definition of the semantics.\(^3\)

### 3.2 Semantics of Support Expressions: Design Decisions

In this section, we discuss informally the design decisions behind the semantics of the support operator $Sp$ in our logic and give an example for the support of an inductive definition. The formal conditions that the supports should satisfy are stated in the equations in Figure 2 and explained in Section 3.3.

The first decision is for every formula $\phi$ to have a unique support, where the support captures the subdomain of mutable functions on which the truthhood of $\phi$ depends. We then define $Sp(\phi)$ such that it evaluates to the support.

The choice for supports of atomic formulas are relatively clear. An atomic formula of the kind $f(x) = y$, where $x$ is of the foreground sort and $f$ is a mutable function, has as its support the singleton set containing the location interpreted for $x$. Atomic formulas that do not involve mutable functions over the foreground have an empty support. Supports for terms can also be similarly defined. The support of a conjunction $\alpha \land \beta$ should clearly be the union of the supports of the two formulas.

**Remark 1.** In traditional separation logic, each pointer field is stored in a separate location using integer offsets. However, in our work, we view pointers as references and disallow pointer arithmetic. A more accurate heaplet for such references can be obtained by taking heaplet to be the pair $(x, f)$ (see Reference [55]), capturing the fact that the formula depends only on the field $f$ of $x$. Such accurate heaplets can be captured in FL as well—we can introduce a non-mutable “field lookup” pointer $L_f$ and use $x.L_f.f$ in programs instead of $x.f$. Supports will then contain elements of the form $L_f(x)$, which is equivalent to $(x, f)$.

**Disjunctions and Conditional Formulas.** What should the support of a formula $\alpha \lor \beta$ be? The choice we make here is that its support is the union of the supports of $\alpha$ and $\beta$. Note that in a model where $\alpha$ is true and $\beta$ is false, we still include the heaplet of $\beta$ in $Sp(\alpha \lor \beta)$. In a sense, this is an overapproximation of the support as far as frame reasoning goes, as surely preserving the model’s definitions on the support of $\alpha$ will preserve the truth of $\alpha$, and hence of $\alpha \lor \beta$.

However, we prefer the support to be the union of the supports of $\alpha$ and $\beta$. We think of the support as the subdomain of the universe that determines the meaning of the formula, whether it be true or false. Consequently, we would like the support of a formula and its negation to be the same. Given that the support of the negation of a disjunction, being a conjunction, is the union of the supports of $\alpha$ and $\beta$, we would like this to be the support for the disjunction as well.

Separation logic makes a different design decision. Logical formulas are not associated with tight supports, rather, the semantics of formulas is defined for models with respect to supports/heaplets, defining whether a given heaplet supports the truthhood of a formula (and not its falsehood). For example, in separation logic, the heaplets that satisfy $f(x) \neq y$ in a model would be all heaplets.

\(^3\)A sufficient restriction is to ensure that formulas of the form $R(t_1, \ldots, t_n)$ (for $R \in I$) do not contain support expressions as subterms.
where the location of \( x \) is not present. This does not coincide with the notion we have chosen for supports. However, for positive formulas, separation logic can handle supports more accurately as it can associate several supports for a formula, yielding two heaplets for formulas of the form \( \alpha \lor \beta \) when they are both true in a model. The decision to have a single support for a formula compels us to define the support of a disjunction to be the union of the supports.

However, sometimes it is possible that for a disjunction \( \alpha \lor \beta \), only one of the disjuncts can possibly be true. In this case, we would like the support of the formula to be the support of the disjunct that is true. We therefore introduce new syntax of the form \( \text{ite}(\gamma : \alpha, \beta) \) in frame logic (standing for if-then-else formulas), whose heaplet is the union of the supports of \( \gamma \) and \( \alpha \) if \( \gamma \) is true, and the supports of \( \gamma \) and \( \beta \) if \( \gamma \) is false. While the truthhood of \( \text{ite}(\gamma : \alpha, \beta) \) as subterms.

\[
\alpha \lor \neg \beta = (\gamma \lor \alpha) \lor (\neg \gamma \lor \beta),
\]

its support is potentially smaller, allowing us to write formulas with tighter supports to support better frame reasoning. Note that the supports of \( \text{ite}(\gamma : \alpha, \beta) \) and its negation \( \text{ite}(\gamma : \neg \alpha, \neg \beta) \) are the same, as we desire.

**Quantifiers.** Turning to quantification, the support for a formula of the form \( \exists x. \alpha \) is hard to define, as its truthhood could depend on the entire universe. We hence provide a mechanism for guarded quantification of the form \( \exists x : \gamma. \alpha \). The semantics of this formula is that there exists some location that satisfies the guard \( \gamma \), for which \( \alpha \) holds. The support for this formula includes the supports of the guard for every interpretation of \( x \), but only the supports of \( \alpha \) when \( x \) is interpreted to be a location that satisfies \( \gamma \). For example, the support of \( \exists x : (x = f(y)). g(x) = z \) only contains the locations interpreted for \( y \) and \( f(y) \), since the support of \( x = f(y) \) is always the location interpreted for \( y \) for every interpretation of \( x \), and we only consider the support of \( g(x) = z \) when \( x \) is equal to \( f(y) \).

Similarly, the support of a guarded universal formula \( \forall x : \gamma. \alpha \) is also the union of the supports of the guard \( \gamma \) for every interpretation of \( x \), along with the supports of \( \alpha \) for every interpretation of \( x \) that satisfies \( \gamma \). Note that although \( \forall x : \gamma. \alpha \) has the same interpretation as \( \forall x. \gamma \implies \alpha \), its support is more carefully defined, in particular, excluding the support of \( \alpha \) for those interpretations of \( x \) that falsify \( \gamma \). The reader may also infer the support of \( \forall x. \gamma \implies \alpha \) by rewriting it as \( \neg(\exists x : \gamma. \neg \alpha) \) and using the fact that negation does not change the support of FL formulas.

**Recursive Definitions.** The support of a formula \( R(\overline{t}) \) for an inductive relation \( R \) defined by

\[
R(\overline{x}) := \rho_R(\overline{x})
\]

not only contains the supports of all the terms in \( \overline{t} \), but also descends into the definition of \( R \), assigning the parameters \( \overline{x} \) in the definition to the actual arguments \( \overline{t} \).

Recursive definitions are designed such that the computation of supports is not dependent on the interpretation of inductive relations. The equations in Figure 2 mainly depend on the syntactic structure of formulas and terms. Only the semantics of guards and the semantics of subterms under a mutable function symbol play a role. For this reason, we disallow guards to contain recursively defined relations or support expressions. We also require that the only functions involving the sort \( S_{\overline{t}} \) are the standard set operations. Thus, mutable functions cannot contain support expressions (which are of sort \( S_{\overline{t}} \)) as subterms.

These restrictions ensure that there indeed exists a unique simultaneous least solution of the equations for the inductive relations and the support expressions, yielding well-defined supports for all frame logic formulas. We now provide an example.

**Example 3.1.** Consider the definition of a predicate \( \text{tree}(x) \) with two unary mutable functions left and right:

\[
\text{tree}(x) := \text{ite}(x = \text{nil} : \top, \alpha) \text{ where}
\]

\[
\alpha = \exists \ell, r : (\ell = \text{left}(x) \land r = \text{right}(x)). \text{tree}(\ell) \land \text{tree}(r) \\
\land \text{Sp}(\text{tree}(\ell)) \cap \text{Sp}(\text{tree}(r)) = \emptyset \land \neg (x \in \text{Sp}(\text{tree}(\ell)) \cup \text{Sp}(\text{tree}(r)))
\]
This defines binary trees with pointer fields \textit{left} and \textit{right}, stating that \(x\) points to a tree if \(x\) is either equal to \textit{nil} (in this case its support is empty) or \(x\) points to subtrees \textit{left}(\(x\)) and \textit{right}(\(x\)) with disjoint supports. The last conjunct says that \(x\) does not belong to the support of the left and right subtrees.

4\footnote{The condition stating disjoint subtrees is, strictly speaking, not required to define trees (under least fixpoint semantics) as it is already ensured by the least fixpoint interpretation.} Access to the support of formulas eases stating disjointness of heaplets, like in separation logic. Following our semantics for \(\text{Sp}\), the support of \textit{tree}(\(x\)) turns out to be precisely the set of locations that are reachable from \(x\) using \textit{left} and \textit{right} pointers, as one would desire.

\subsection{Formal Semantics of Frame Logic}

In this section, we present the formal semantics of frame logic, focusing on the semantics of support expressions and inductive definitions. To do this, we first introduce a semantics that treats support expressions and the symbols in \(I\) as uninterpreted. We refer to this semantics as \textit{uninterpreted semantics}. We introduce some terminology below.

An occurrence of a variable \(x\) in a formula is free if it does not occur under the scope of a quantifier for \(x\). By renaming variables, we can assume that each variable only occurs freely in a formula or is quantified by exactly one quantifier in the formula. We write \(\varphi(x_1, \ldots, x_k)\) to indicate that the free variables of \(\varphi\) are among \(x_1, \ldots, x_k\). Substitution of a term \(t\) for all free occurrences of variable \(x\) in a formula \(\varphi\) is denoted \(\varphi[t/x]\). Multiple variables are substituted simultaneously as \(\varphi[t_1/x_1, \ldots, t_n/x_n]\). We abbreviate this by \(\varphi[t/x]\).

A model is of the form \(M = (U; [\cdot]_M)\) where \(U = (U_\tau)_{\tau \in S}\) contains a universe for each sort, and an interpretation function \([\cdot]_M\). The universe for the sort \(\sigma_{S(f)}\) is the power set of the universe for \(\sigma_f\).

A variable assignment is a function \(\nu\) that assigns to each variable a concrete element from the universe for the sort of the variable. For a variable \(x\), we write \(D_x\) for the universe of the sort of
x (the domain of x). For a variable x and an element u ∈ Dx, we write ν[x ← u] for the variable assignment that is obtained from ν by changing the value assigned for x to u.

The interpretation function [.]M maps each constant c of sort τ to an element [c]M ∈ Uτ, each function symbol f : τ1 × ⋯ × τm → τ to a concrete function [f]M : Uτ1 × ⋯ × Uτm → Uτ, and each relation symbol R ∈ R ∪ I of type τ1 × ⋯ × τm to a concrete relation [R]M ⊆ Uτ1 × ⋯ × Uτm. These interpretations are assumed to satisfy the background theories (see Section 2). Furthermore, the interpretation function maps each expression of the form Sp(φ) to a function [Sp(φ)]M that assigns to each variable assignment ν a set [Sp(φ)]M(ν) of foreground elements. The set [Sp(φ)]M(ν) corresponds to the support of the formula when the free variables are interpreted by ν. Similarly, [Sp(t)]M is a function from variable assignments to sets of foreground elements.

Based on such models, we can define the semantics of terms and formulas in the standard way. The only construct that is non-standard in our logic are terms of the form Sp(φ), for which the semantics is directly given by the interpretation function in Figure 2. We write [t]M,ν for the interpretation of a term t in M with variable assignment ν. With this convention, [Sp(φ)]M(ν) denotes the same thing as [Sp(φ)]M,ν. As usual, we write M, ν ⊨ φ to indicate that the formula φ is true in M with the free variables interpreted by ν, and [φ]M denotes the relation defined by the formula φ with free variables X. Note that the support of ∀y : ν. φ is the same as the support of ∃y : ν. φ, since ∀y : ν. φ is a shorthand for ¬(∃y : ν. ¬φ).

We refer to the above semantics as the uninterpreted semantics of φ, because we do not give a specific meaning to inductive definitions and support expressions.

Now, let us define the true semantics for FEL. The relation symbols R ∈ I represent inductively defined relations, which are defined by equations of the form R(X) := ρR(X) (see Figure 1). In the intended meaning, R is interpreted as the least relation that satisfies the equation

[R(X)]M = [ρR(X)]M.

The usual requirement for the existence of a unique least fixpoint of the equation is that the definition of R does not negatively depend on R. For this reason, we require that in ρR(X) each occurrence of an inductive predicate R′ ∈ I is either inside a support expression or it occurs under an even number of negations.5

Given a model, every support expression is evaluated to a set of foreground elements in the model (under a given variable assignment ν). Formally, we are interested in models in which the support expressions are interpreted to be the sets that correspond to the smallest solution of the equations given in Figure 2. The intuition behind these definitions was explained in Section 3.2.

Example 3.2. Consider the inductive definition tree(x) from Example 3.1. We illustrate that the equations from Figure 2 indeed yield the desired value for Sp(tree(x)). First, note that the supports Sp(x = nil) = Sp(x = ⊤) = 0. Denoting by ψ[u] a variable assignment that assigns u to the free variable in a formula ψ, observe that Sp(tree(x))[u] = ∅ if u is nil, and Sp(tree(x))[u] = Sp(α)[u] if u ≠ nil. The formula α is existentially quantified with guard ℓ = left(x) ∧ r = right(x). The support of the guard is {u}, because mutable functions are applied to x, whose value is u. The support of the body of the quantified expression α is the union of the supports of tree(ℓ)[left(u)] and tree(r)[right(u)] (the assignments for ℓ and r that make the guard true). Therefore, when u ≠ nil, computing the support of tree(x)[u] according to Figure 2 adds u to the set and continues to recurse on the subtrees of u, as desired.

A frame model is a model in which the interpretation of the inductive relations and of the support expressions corresponds to the least solution of the respective equations. We formalize this idea and prove the following proposition in Section 3.4.

---

5As usual, it would be sufficient to forbid negative occurrences of inductive predicates in mutual recursion.
For each model \( \hat{M} \) only differ in the interpretation be a pre-model, \( M \) spans a \( \in \) \( = \) \( \leq \) \( < \) \( \subseteq \) \( \subseteq \) \( \sqsubseteq \) \( \sqsubseteq \) \( \text{if} \) \( \text{does not satisfy the support equations, and} \) \( \text{is defined like a model with the differences that a pre-model does not interpret the} \) \( \text{for all support expressions, and} \) \( \text{and} \) \( \text{for all inductive relations} \) \( \sigma \) \( \hat{M} \) \( \cap \) \( \leq \) \( \llbracket \) \( \text{is a partial order.} \) \( \text{as well as} \) \( \text{for the inductive definitions as the} \) \( \in \) \( \in \) \( \sigma \) \( \hat{M} \) \( \text{such that there does not exist an element in the intersection} \) \( \text{and} \) \( \text{the least fixpoint for the inductive predicates.} \) 

In the following, we refer to the equations for the support expressions from Figure 2 as support equations, and to the equations \( \llbracket R(\overline{x}) \rrbracket_M = \llbracket p_R(\overline{x}) \rrbracket_M \) for the inductive definitions as the inductive equations.

For \( M_1, M_2 \in \text{Mod}(\hat{M}) \), we let \( M_1 \leq_f M_2 \) if

- \( \llbracket \text{Sp}(\varphi) \rrbracket_{M_1}(\nu) \subseteq \llbracket \text{Sp}(\varphi) \rrbracket_{M_2}(\nu) \) as well as \( \llbracket \text{Sp}(t) \rrbracket_{M_1}(\nu) \subseteq \llbracket \text{Sp}(t) \rrbracket_{M_2}(\nu) \) for all support expressions and all variable assignments \( \nu \).

Note that \( \leq_f \) is not a partial order but only a preorder: For two models \( M_1, M_2 \) that differ only in their interpretations of the inductive relations, we have \( M_1 \leq_f M_2 \) and \( M_2 \leq_f M_1 \). We write \( M_1 <_f M_2 \) if \( M_1 \leq_f M_2 \) and not \( M_2 \leq_f M_1 \).

We further define \( M_1 \leq_s M_2 \) if

- \( \llbracket \text{Sp}(\varphi) \rrbracket_{M_1} = \llbracket \text{Sp}(\varphi) \rrbracket_{M_2} \) as well as \( \llbracket \text{Sp}(t) \rrbracket_{M_1} = \llbracket \text{Sp}(t) \rrbracket_{M_2} \) for all support expressions, and
- \( \llbracket I \rrbracket_{M_1} \subseteq \llbracket I \rrbracket_{M_2} \) for all inductive relations \( I \in \mathcal{I} \).

The relation \( \leq_s \) is a partial order.

We say that \( M \in \text{Mod}(\hat{M}) \) is a frame model if its interpretation function \( \llbracket \cdot \rrbracket_M \) satisfies the inductive equations and the support equations, and furthermore

(1) each \( M' \in \text{Mod}(\hat{M}) \) with \( M' <_f M \) does not satisfy the support equations, and
(2) each \( M' \in \text{Mod}(\hat{M}) \) with \( M' <_s M \) does not satisfy the inductive equations.

For proving the existence of a unique frame model, we use the following lemma for dealing with guards and terms with mutable functions:

**Lemma 3.4.** Let \( \hat{M} \) be a pre-model, \( M_1, M_2 \in \text{Mod}(\hat{M}) \), and \( \nu \) be a variable assignment.

(1) If \( \varphi \) is a formula that does not use inductive relations and support expressions, then \( M_1, \nu \models \varphi \) if and only if \( M_2, \nu \models \varphi \).

(2) If \( t \) is a term that has no support expressions as subterms, then \( \llbracket t \rrbracket_{M_1, \nu} = \llbracket t \rrbracket_{M_2, \nu} \).

(3) If \( t = f(t_1, \ldots, t_n) \) is a term with a mutable function symbol \( f \in F_m \), then \( \llbracket t_i \rrbracket_{M_1, \nu} = \llbracket t_i \rrbracket_{M_2, \nu} \) for all \( i \).

**Proof.** Parts 1 and 2 are immediate from the fact that \( M_1 \) and \( M_2 \) only differ in the interpretation of the inductive relations and support expressions. For the third claim, note that we assumed that the only functions involving arguments of sort \( \sigma_S(f) \) are the standard functions for set manipulation.
Hence, a term build from a mutable function symbol cannot have support expressions as subterms. Therefore, the third claim follows from the second one.

We now prove Proposition 3.3 (see Section 3.3). We first restate the proposition formally using the terminology developed above.

**Proposition 3.5.** For each pre-model \( \hat{M} \), there is a unique frame model in \( \text{Mod}(\hat{M}) \).

**Proof.** The support equations define an operator \( \mu_f \) on \( \text{Mod}(\hat{M}) \). This operator \( \mu_f \) is defined in a standard way, as explained in the following: Let \( M \in \text{Mod}(\hat{M}) \). Then, \( \mu_f(M) \) is a model in \( \text{Mod}(\hat{M}) \) where \( \llbracket \text{Sp}(\varphi) \rrbracket_{\mu_f(M)} \), respectively, \( \llbracket \text{Sp}(t) \rrbracket_{\mu_f(M)} \), is obtained by taking the right-hand side of the corresponding equation. For example, \( \llbracket \text{Sp}(\varphi_1 \land \varphi_2) \rrbracket_{\mu_f(M)}(v) = \llbracket \text{Sp}(\varphi_1) \rrbracket_M(v) \cup \llbracket \text{Sp}(\varphi_2) \rrbracket_M(v) \). The interpretation of the inductive predicates is left unchanged by \( \mu_f \).

We can show that \( \mu_f \) is a monotonic operator on \( (\text{Mod}(\hat{M}), \leq_f) \), that is, for all \( M_1, M_2 \in \text{Mod}(\hat{M}) \) with \( M_1 \leq_f M_2 \), we have that \( \mu_f(M_1) \leq \mu_f(M_2) \). It is routine to check monotonicity of \( \mu_f \) by induction on the structure of the support expressions. We use Lemma 3.4 for the only cases in which the semantics of formulas and terms is used in the support equations, namely, \( \text{ite} \)-formulas, existential formulas, and terms \( f(t_1, \ldots, t_n) \) with mutable function \( f \). Consider, for example, the support equation
\[
\llbracket \text{Sp}(f(t_1, \ldots, t_n)) \rrbracket_{\mu_f(M)}(v) = \bigcup_{i \text{ with } t_i \text{ of sort } \sigma_i} \llbracket t_i \rrbracket_{M, v} \cup \bigcup_{i=1}^n \llbracket \text{Sp}(t_i) \rrbracket_{M}(v)
\]
for \( f \in F_m \), and let \( M_1 \leq_f M_2 \) be in \( \text{Mod}(\hat{M}) \) and \( v \) be a variable assignment. Then
\[
\llbracket \text{Sp}(f(t_1, \ldots, t_n)) \rrbracket_{\mu_f(M_1)}(v) = \bigcup_{i \text{ with } t_i \text{ of sort } \sigma_i} \llbracket t_i \rrbracket_{M_1, v} \cup \bigcup_{i=1}^n \llbracket \text{Sp}(t_i) \rrbracket_{M_1}(v)
\]
\[
\quad = \bigcup_{i \text{ with } t_i \text{ of sort } \sigma_i} \llbracket t_i \rrbracket_{M_2, v} \cup \bigcup_{i=1}^n \llbracket \text{Sp}(t_i) \rrbracket_{M_2}(v)
\]
\[
\quad = \llbracket \text{Sp}(f(t_1, \ldots, t_n)) \rrbracket_{\mu_f(M_2)}(v),
\]
where (1) holds because of Lemma 3.4, and (2) holds because \( M_1 \leq_f M_2 \).

As a further case, consider the support equation for \( R(\overline{t}) \), where \( R \) is an inductively defined relation and \( t = (t_1, \ldots, t_n) \).
\[
\llbracket \text{Sp}(R(\overline{t})) \rrbracket_{\mu_f(M_1)}(v) = \llbracket \text{Sp}(\rho_R(\overline{x})) \rrbracket_{M_1}(v[\overline{x} \leftarrow [\overline{t}]_{M_1, v}]) \cup \bigcup_{i=1}^n \llbracket \text{Sp}(t_i) \rrbracket_{M_1}(v)
\]
\[
\quad \subseteq \llbracket \text{Sp}(\rho_R(\overline{x})) \rrbracket_{M_2}(v[\overline{x} \leftarrow [\overline{t}]_{M_2, v}]) \cup \bigcup_{i=1}^n \llbracket \text{Sp}(t_i) \rrbracket_{M_2}(v) = \llbracket \text{Sp}(R(\overline{t})) \rrbracket_{\mu_f(M_2)}(v)
\]
For the inclusion (*), we use the fact that the \( t_i \) do not contain support expressions as subterms by our restriction of the type of inductively defined relations. Hence, by Lemma 3.4, \( \llbracket t_i \rrbracket_{M_1, v} = \llbracket t_i \rrbracket_{M_2, v} \). Similarly, one can show the inclusion for the other support equations.

We also obtain an operator \( \mu_i \) from the inductive equations, which leaves the interpretation of the support expressions unchanged. The operator \( \mu_i \) is monotonic on \( (\text{Mod}(\hat{M}), \leq_i) \) because inductive predicates can only be used positively in the inductive definitions, and furthermore \( \leq_i \) only compares models with the same interpretation of the support expressions.
To obtain the unique frame model, we first consider the subset of $\text{Mod}(\hat{M})$ in which all inductive predicates are interpreted as empty set. On this set of models, $\leq_f$ is a partial order and forms a complete lattice (the join and meet for the lattice are obtained by taking the point-wise union, respectively, intersection, of the interpretations of the support expression). By the Knaster-Tarski theorem [69], there is a unique least fixpoint of $\mu_f$. This fixpoint can be obtained by iterating $\mu_f$ starting from the model in $\text{Mod}(\hat{M})$ that interprets all inductive relations and the support expression by the empty set (in general, this iteration is over the ordinal numbers, not just the natural numbers). Let $M_f$ be this least fixpoint.

The subset of $\text{Mod}(\hat{M})$ in which the support expressions are interpreted as in $M_f$ forms a complete lattice with the partial order $\leq_i$. Again by the Knaster-Tarski Theorem, there is a unique least fixpoint. This least fixpoint can be obtained by iterating the operator $\mu_i$ starting from $M_f$ (again, the iteration is over the ordinals).

Denote the resulting model by $M_{f,i}$. It interprets the support expressions in the same way as $M_f$, and thus $M_f \leq_i M_{f,i}$ and $M_{f,i} \leq_i M_i$. By monotonicity of $\mu_i$, $M_{f,i}$ is also a fixpoint of $\mu_i$ and thus satisfies the support equations. Hence $M_{f,i}$ satisfies the inductive equations and the support equations. It can easily be checked that $M_{f,i}$ also satisfies the other conditions of a frame model: Let $M \in \text{Mod}(\hat{M})$ with $M \prec_i M_{f,i}$. Then, also $M \leq_i M_f$ and assuming that $M$ satisfies the support equations yields a smaller fixpoint of $\mu_i$, and thus a contradiction. Similarly, a model $M \prec_i M_{f,i}$ cannot satisfy the inductive equations.

It follows that $M_{f,i}$ is a frame model in $\text{Mod}(\hat{M})$. Uniqueness follows from the uniqueness of the least fixpoints of $\mu_f$ and $\mu_i$ as used in the construction of $M_{f,i}$.

\section*{Consequences of Frame Logic Semantics}

We now highlight some consequences of the semantics of FL and highlight differences with other logics.

\textbf{Global Heap Semantics vs. Local Heap Semantics:} Our semantics is \textit{global} in the sense that formulas are interpreted on the entire model, similar to FOL. Therefore, for example, tautologies in first-order logic such as $\alpha \implies (\alpha \land \top)$ and $\alpha \implies (\alpha \lor \bot)$ are also valid in FL. Supports of formulas capture the part of a model on which their truth depends and roughly correspond to heaplets in separation logic.

In contrast, Separation Logic itself has a “local heap” semantics, which is defined not on the global heap but with respect to local heaplets. The semantics of SL is captured by rules determining the truth value of expressions of the form $s, h \models \phi$, where $h$ is a heaplet [63].

An example of the above distinction is the FL formula $\exists y : y = f(x)$. $\top$. This formula is valid in FL but the “corresponding” SL formula $\exists y. x \mapsto f y$ is not valid: It only holds on the singleton heaplet $h_x$ containing the valuation of $x$ (which is a heap location), but not on other heaplets. Note that $Sp(\exists y : y = f(x)). \top$ is this heaplet $h_x$.

\textbf{Substitutions:} Another interesting consequence of FL semantics is that substituting a formula with another equivalent formula (in terms of truthhood) may not result in equivalent formulas.

Consider the formulas $\top$ and $f(x) = f(x)$ for a mutable function $f$. These are equivalent in FL in terms of truthhood (they are both valid, holding in all models). Consider the valid formula $Sp(f(x) = f(x)) = \{x\}$. Substituting $f(x) = f(x)$ with $\top$ will result in $Sp(\top) = \{x\}$, which is invalid, since $Sp(\top) = \emptyset$.

More precisely, there can be formulas that are equivalent in terms of truthhood (in the global heap) but have different supports, and hence one may not be substituted by another in every context. However, it is true that if $\alpha$ and $\beta$ are equivalent in terms of truthhood \textit{and} have the
same supports in every model, then they can always be substituted for each other. This motivates a stronger definition of equivalence for FL formulas:

**Definition 3.6 (Equivalence of FL Formulas).** FL formulas \( \varphi_1 \) and \( \varphi_2 \) are said to be equivalent, denoted \( \varphi_1 \equiv \varphi_2 \), if for every frame model \( M \) and interpretation of free variables \( \nu, M, \nu \vDash \varphi_1 \) iff \( M, \nu \vDash \varphi_2 \) and \( \llbracket \text{Sp}(\varphi_1) \rrbracket_{M, \nu} = \llbracket \text{Sp}(\varphi_2) \rrbracket_{M, \nu} \).

Formulas are not equivalent if they merely imply each other; it must also be the case that their supports are equal. Reasoning directly in FL requires care to cater the above notion. However, as we point out in Section 6.2, we can convert FL formulas to FO-RD and use existing reasoning techniques for FO-RD (which admits substitutions in the familiar sense, i.e., between formulas implied by each other).

**Supports and Heaplets:** The design decision in FL is for every formula to have precisely one support. In SL, there can be several heaplets under which a formula can hold true. For example, the formula \( \top \) holds in *any* heaplet in SL, while we have chosen the support of \( \top \) (and that of \( \bot \)) to be \( \emptyset \). However, these unique supports only affect the truth value of a formula if they are referred to using \( \text{Sp} \) expressions. For example, \( \alpha \land \neg (\alpha) \implies \bot \) and \( \alpha \lor \neg \alpha \implies \top \) are valid in FL, despite the fact that the support of the antecedent and that of the consequent is different (this property does not hold for SL formulas in general).

### 3.5 A Frame Theorem

The support of a formula can be used for frame reasoning in the following sense: If we modify a model \( M \) by changing the interpretation of the mutable functions (e.g., a program modifying pointers), then truth values of formulas do not change if the change happens outside the support of the formula. This is formalized and proven below.

Given two models \( M, M' \) over the same universe, we say that \( M' \) is a mutation of \( M \) if \( \llbracket R \rrbracket_{M} = \llbracket R \rrbracket_{M'} \), \( c_{M} = c_{M'} \), and \( \llbracket f \rrbracket_{M} = \llbracket f \rrbracket_{M'} \), for all constants \( c \), relations \( R \in \mathcal{R} \), and functions \( f \in F \setminus F_{m} \). In other words, \( M \) can only be different from \( M' \) on the interpretations of the mutable functions, the inductive relations, and the support expressions.

Given a subset \( X \subseteq U_{m} \) of the elements from the foreground universe, we say that the *mutation is stable on \( X \)* if the values of the mutable functions did not change on arguments from \( X \), that is, \( \llbracket f \rrbracket_{M}(u_{1}, \ldots, u_{n}) = \llbracket f \rrbracket_{M'}(u_{1}, \ldots, u_{n}) \) for all mutable functions \( f \in F_{m} \) and all appropriate tuples \( u_{1}, \ldots, u_{n} \) of arguments with \( \{ u_{1}, \ldots, u_{n} \} \cap X \neq \emptyset \).

**Theorem 3.7 (Frame Theorem).** Let \( M, M' \) be frame models such that \( M' \) is a mutation of \( M \) that is stable on \( X \subseteq U_{m} \), and let \( \nu \) be a variable assignment. Then, \( M, \nu \vDash \alpha \) iff \( M', \nu \vDash \alpha \) for all formulas \( \alpha \) with \( \llbracket \text{Sp}(\alpha) \rrbracket_{M}(\nu) \subseteq X \), and \( \llbracket t \rrbracket_{M, \nu} = \llbracket t \rrbracket_{M', \nu} \) for all terms \( t \) with \( \llbracket \text{Sp}(t) \rrbracket_{M}(\nu) \subseteq X \).

**Proof.** The intuition behind the statement of the theorem should be clear. The support of a formula/term contains the elements on which mutable functions are dereferenced to evaluate the formula/term. If the mutable functions do not change on this set, then the evaluation does not change.

For a formal proof of the Frame Theorem, we refer to the terminology and definitions introduced in Section 3.4 and to the proof of Proposition 3.5 in Section 3.4, in which the unique frame model is obtained by iterating the operators \( \mu_{f} \) and \( \mu_{i} \), which are defined by the support equations and the inductive equations.

In general, this iteration of the operators ranges over ordinals (not just natural numbers). For an ordinal \( \eta \), let \( M_{\eta} \) and \( M'_{\eta} \) be the models at step \( \eta \) of the fixpoint iteration for obtaining the frame models \( M \) and \( M' \). So, the sequence of the \( M_{\eta} \) have monotonically increasing interpretations of the inductive relations and support expressions and are equal to \( M \) on the interpretation of the other relations and functions. The frame model \( M \) is obtained at some stage \( \xi \) of the fixpoint iteration,
so $M = M_\xi$. More precisely, the frame model is constructed by first iterating the operator $\mu_i$ until the fixpoint of the support expressions is reached. During this iteration, the inductive relations are interpreted as empty. Then, the operator $\mu_i$ is iterated until also the inductive relations reach their fixpoint. Below, we do an induction on $\eta$. In that induction, we do not explicitly distinguish these two phases, because it does not play any role for the arguments (only in one place, and we mention it explicitly there).

By induction on $\eta$, we can show that $M_\eta, v \models \varphi \iff M_\eta', v \models \varphi$, and $[t]_{M_\eta, v} = [t]_{M_\eta', v}$ for all variable assignments $v$ and all formulas $\varphi$ with $[Sp(\varphi)]_{M}(v) \subseteq X$, respectively, terms $t$ with $[Sp(t)]_{M}(v) \subseteq X$. For each $\eta$, we furthermore do an induction on the structure of the formulas, respectively, terms.

Note that the assumption that the support is contained in $X$ refers to the support in $M$. So, when applying the induction, we have to verify that the condition on the support of a formula/term is satisfied in $M$ (and not in $M_\eta$).

For the formulas, the induction is straightforward, using Lemma 3.4 in the cases of existential formulas and $ite$-formulas. Consider, for example, the case of an existential formula $\psi = \exists y : \gamma \cdot \varphi$ with $[Sp(\psi)]_{M} \subseteq X$.

\[
M_\eta, v \models \exists y : \gamma \cdot \varphi \\
\iff \exists u \in D_y : M_\eta, v[y \leftarrow u] \models \gamma \\
\quad \text{and } M_\eta, v[y \leftarrow u] \models \varphi \\
\iff \exists u \in D_y : M_\eta', v[y \leftarrow u] \models \gamma \\
\quad \text{and } M_\eta', v[y \leftarrow u] \models \varphi \\
\iff M_\eta', v \models \exists y : \gamma \cdot \varphi,
\]

where $(*)$ holds by induction on the structure of the formula. We only have to verify that $[Sp(\gamma)]_{M}(v[y \leftarrow u]) \subseteq X$ and $[Sp(\varphi)]_{M}(v[y \leftarrow u]) \subseteq X$ to use the induction hypothesis.

Since $\gamma$ is a guard of an existential formula, it satisfies the condition of Lemma 3.4, and therefore its truth value is the same in $(M_\eta, v[y \leftarrow u])$ for all ordinals $\eta$ (Lemma 3.4 applies, because all the models $M_\eta$ differ only in the interpretations of the support expressions and inductive relations, and thus have the same pre-model). In particular, $M, v[y \leftarrow u] \models \gamma$, since $M = M_\xi$ for some ordinal $\xi$. From the equations for the supports, we obtain $[Sp(\gamma)]_{M}(v[y \leftarrow u]) \subseteq [Sp(\psi)]_{M}(v)$ and $[Sp(\varphi)]_{M}(v[y \leftarrow u]) \subseteq [Sp(\psi)]_{M}(v)$. The desired claim now follows from the fact that $[Sp(\psi)]_{M}(v) \subseteq X$.

For inductive relations $R$ with definition $R(\bar{x}) := \rho_R(\bar{x})$, we have to use the induction on the ordinal $\eta$. Assume that $\varphi = R(\bar{t})$ for $\bar{t} = (t_1, \ldots, t_n)$ and that $[Sp(\varphi)]_{M}(v) \subseteq X$. Then, $[Sp(\rho_R(\bar{x}))]_{M}(v[\bar{x} \leftarrow \bar{t}]) \subseteq X$ and $[Sp(t_i)]_{M}(v) \subseteq X$ for all $i$ by the support equations.

For the case of a limit ordinal $\eta$, the inductive relations of $M_\eta$, respectively, $M_\eta'$, are obtained by taking the union of the interpretations of the inductive relations for all $M_\xi$, respectively, $M_\xi'$, for all $\xi < \eta$. So, the claim follows directly by induction.

For a successor ordinal $\eta + 1$, we can assume that we are in the second phase of the construction of the frame model (the iteration of the operator $\mu_i$). For the first phase the claim trivially holds, because all the inductive relations are interpreted as empty. Thus, we have

\[
M_{\eta+1}, v \models R(\bar{t}) \\
\iff M_\eta, v \models \rho_R(\bar{t}) \\
\iff M_\eta, v[\bar{x} \leftarrow [\bar{t}]_{M_\eta, v}] \models \rho_R(\bar{x}) \\
\quad \text{(*)} \\
\iff M'_\eta, v[\bar{x} \leftarrow [\bar{t}]_{M'_\eta, v}] \models \rho_R(\bar{x}) \\
\iff M'_{\eta+1}, v \models R(\bar{t}),
\]

where $(*)$ holds by induction on $\eta$.
The other cases for formulas are similar (or simpler).

Concerning the terms, we also present some cases only, the other cases being similar or simpler.

We start with the case \( t = f(t_1, \ldots, t_n) \) for a mutable function \( f \). Let \( \nu \) be a variable assignment with \([ t ]_{\nu}^{M} \subseteq X\). By the support equations, \([ t_i ]_{\nu}^{M} \subseteq X\) for all \( i \). We have \([ t ]_{\nu}^{M} = \bigcup f([ t_1 ]_{\nu}^{M}, \ldots, [ t_n ]_{\nu}^{M})\). By induction on the structure of terms, we have \([ t ]_{\nu}^{M} = [ t ]_{\nu}^{M} = [ t ]_{\nu}^{M} =: u_t\). By Lemma 3.4, we conclude that \([ t ]_{\nu}^{M} \subseteq [ t ]_{\nu}^{M}\). Since \( f \) is mutable, it contains at least one argument of sort \( \sigma_i \), say, \( t_j \). Then, \([ t_j ]_{\nu}^{M} \subseteq [ Sp(t_j) ]_{\nu}^{M} \subseteq X\), and the mutation did not change the function value of \( f \) on the tuple \((u_1, \ldots, u_n)\). So, we obtain in summary that \([ t ]_{\nu}^{M} \subseteq [ f ]_{\nu}^{M}(u_1, \ldots, u_n)\).

Now, consider terms of the form \( Sp(\phi)\). We need to proceed by induction on the structure of \( \phi \).

We present the case of \( \phi = \text{ite}(\gamma : \phi_1, \phi_2)\). Let \( \nu \) be a variable assignment with \([ Sp(\phi) ]_{\nu}^{M} \subseteq X\). Assume that \( M_{\nu}, \nu \models \gamma \). By the condition on guards, Lemma 3.4 yields that \( M, \nu \models \gamma \) and thus \([ Sp(\phi) ]_{\nu}^{M} \subseteq X\) and \([ Sp(\phi_1) ]_{\nu}^{M} \subseteq X\). We obtain
\[
[ Sp(\phi) ]_{\nu}^{M} (v) = [ Sp(\gamma) ]_{\nu}^{M} (v) \cup [ Sp(\phi_1) ]_{\nu}^{M} (v)
\]
\[
= [ Sp(\gamma) ]_{\nu}^{M} (v) \cup [ Sp(\phi_1) ]_{\nu}^{M} (v)
\]
\[
= [ Sp(\phi) ]_{\nu}^{M} (v),
\]
where \( (\ast) \) follows by induction on the structure of the formula inside the support expression. The case \( M_{\nu}, v \not\models \gamma \) is analogous.

Now, consider \( Sp(\phi) \) with \( \phi = R(\overline{t}) \) for an inductively defined relation \( R \) with definition \( R(\overline{x}) = \rho_R(\overline{x})\) and \( \overline{t} = (t_1, \ldots, t_n)\). Let \( \nu \) be a variable assignment with \([ Sp(\phi) ]_{\nu}^{M} \subseteq X\). By the support equations, \([ Sp(\rho_R(\overline{x})) ]_{\nu}^{M}(v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M}) \subseteq X\) and \([ Sp(t_i) ]_{\nu}^{M} \subseteq X\).

Let \( \eta + 1 \) be a successor ordinal. Then
\[
[ Sp(\rho_R(\overline{x})) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M}) = \bigcup_{i=1}^{n} [ Sp(t_i) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M})
\]
\[
= [ Sp(\rho_R(\overline{x})) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M}) \cup [ Sp(t_i) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M})
\]
\[
= [ Sp(\rho_R(\overline{x})) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M}) \cup [ Sp(t_i) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M})
\]
\[
= [ Sp(\rho_R(\overline{x})) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M}) \cup [ Sp(t_i) ]_{\nu}^{M} (v[\overline{x} \leftarrow [ \overline{t} ]_{\nu}^{M})
\]
where \( (\ast) \) holds by induction on \( \eta \). We can apply the induction hypothesis, because the terms \( t_i \) do not contain support expressions by the restriction on the type of inductive relations, and thus \([ t ]_{\nu}^{M} = [ t ]_{\nu}^{M} \subseteq [ t ]_{\nu}^{M}\).

The proof of the other cases works in a similar fashion.

\[\square\]

### 3.6 Reduction from Frame Logic to FO-RD

The only extension of frame logic compared to FO-RD is the operator \( Sp \), which defines a function from interpretations of free variables to sets of foreground elements. The semantics of this operator can be captured within FO-RD itself, so reasoning within frame logic can be reduced to reasoning within FO-RD.

A formula \( \alpha(\overline{y}) \) with \( \overline{y} = y_1, \ldots, y_m \) has one support for each interpretation of the free variables. We capture these supports by an inductively defined relation \( Sp_{\alpha}(\overline{y}, z) \) of arity \( m + 1 \) such that for each frame model \( M \), we have \((u_1, \ldots, u_m, u) \in [ Sp_{\alpha} ]_{M} \) if \( u \in [ Sp(\alpha) ]_{M} (v) \) for the interpretation \( v \) that interprets \( y_i \) as \( u_i \).

Since the semantics of \( Sp(\alpha) \) is defined over the structure of \( \alpha \), we introduce corresponding inductively defined relations \( Sp_{\beta} \) and \( Sp_{t} \) for all subformulas \( \beta \) and subterms \( t \) of either \( \alpha \) or of a formula \( \rho_R \) for \( R \in I \).

ACM Transactions on Programming Languages and Systems, Vol. 45, No. 2, Article 7. Publication date: May 2023.
The equations for supports from Figure 2 can be expressed by inductive definitions for the relations \( Sp_\varphi \). The translations are shown in Figure 3. For the definitions, we assume that \( \overline{y} \) contains all variables that are used in \( \alpha \) and in the formulas \( \rho_R \) of the inductive definitions. We further assume that each variable either is used in at most one of the formulas \( \alpha \) or \( \rho_R \), and either only occurs freely in it or is quantified at most once. The relations \( Sp_\varphi \) are all of arity \( m+1 \), even if the subformulas do not use some of the variables. In practice, one would rather use relations of arities as small as possible, referring only to the relevant variables. In a general definition, this is, however, rather cumbersome to write, so we use this simpler version in which we do not have to rearrange and adapt the variables according to their use in subformulas.

For the definition of \( Sp_{R(\overline{t})} \) where \( R \) is an inductively defined relation, note that the variables \( x \) from the definition of \( R \) are contained in \( \overline{y} \) by the above assumptions and are substituted by the terms in \( \overline{t} \) in the first part of the formula. Similarly, the quantified variable \( x \) in an existential formula is contained in \( \overline{y} \).

It is not hard to see that general frame logic formulas can be translated to FO-RD formulas that make use of these new inductively defined relations.

**Proposition 3.8.** For every frame logic formula there is an equi-satisfiable FO-RD formula with the signature extended by auxiliary predicates for recursive definitions of supports.

### 3.7 Expressing Data-structure Properties in FL

We now present the formulation of several data-structures and properties about them in FL. Figure 4 depicts formulations of singly- and doubly-linked lists, list segments, lengths of lists, sorted lists, the multiset of keys stored in a list (assuming a background sort of multisets), binary trees, their heights, and AVL trees. In all these definitions, the support operator plays a crucial role. We also present a formulation of single threaded binary trees (adapted from Reference [13]), which are binary trees where, apart from tree-edges, there is a pointer \( t_{\text{next}} \) that connects every
list(x) := ite(x = nil, T, \exists z : z = next(x). list(z) \wedge x \notin Sp(list(z))) (linked list)

dll(x) := ite(x = nil, T, ite(next(x) = nil, T, \\
\exists z : z = next(x). prev(z) = x \wedge dll(z) \wedge x \notin Sp(dll(z)))) (doubly linked list)

lseg(x, y) := ite(x = y, T, \exists z : z = next(x). lseg(z, y) \wedge x \notin Sp(lseg(z, y))) (linked list segment)

length(x, n) := ite(x = nil, n = 0, \exists z : z = next(x). length(z, n - 1)) (length of list)

slist(x) := ite(x = nil, T, ite(next(x) = nil, T, \\
\exists z : z = next(x). x = key(z) \wedge slist(z) \wedge x \notin Sp(slist(z)))) (sorted list)

mkeys(x, M) := ite(x = nil, M = \emptyset, \exists z : z = next(x). \exists M_2 : M = M_2 \cup_m \{key(x)\}.
\exists M' : M' = M_2 \cup_m \{key(z)\}) (multiset of keys in a linked list)\textsuperscript{6}

btree(x) := ite(x = nil, T, \exists l, r : l = left(x) \wedge r = right(x).

\begin{align*}
\text{btree}(l) \wedge \text{btree}(r) \wedge x \notin Sp(\text{btree}(l)) \wedge x \notin Sp(\text{btree}(r)) \wedge \\
\text{Sp(\text{btree}(l)}) \cap \text{Sp(\text{btree}(r))} = \emptyset
\end{align*}

(minimum key in a tree/dag)

minkey(x, n) := ite(x = nil, n = \infty, \exists l, r, n_l, n_r : l = left(x) \wedge r = right(x).

\begin{align*}
n_l = \text{minkey}(l) \wedge n_r = \text{minkey}(r) \wedge n = \min(n_l, key(x), n_r))
\end{align*}

(maximum key in a tree/dag)

maxkey(x, n) := ite(x = nil, n = -\infty, \exists l, r, n_l, n_r : l = left(x) \wedge r = right(x).

\begin{align*}
n_l = \text{maxkey}(l) \wedge n_r = \text{maxkey}(r) \wedge n = \max(n_l, key(x), n_r))
\end{align*}

(binary tree)

bst(x) := ite(x = nil, T, \exists l, r, maxl, minr : l = left(x) \wedge r = right(x).

\begin{align*}
\text{maxkey}(l, maxl) \wedge \text{minkey}(r, minr) \wedge \\
\text{bst}(l) \wedge \text{bst}(r) \wedge \text{maxl} \leq \text{key}(x) \wedge \text{key}(x) \leq \text{minr} \wedge \\
x \notin Sp(\text{bst}(l)) \wedge x \notin Sp(\text{bst}(r)) \wedge \text{Sp(bst(l)}) \cap \text{Sp(bst(r))} = \emptyset
\end{align*}

(balance factor for an AVL tree)

height(x, n) := ite(x = nil, n = 0, \exists l, r, n_l, n_r : l = left(x) \wedge r = right(x).

\begin{align*}
\text{height}(l, n_l) \wedge \text{height}(r, n_r) \wedge \text{ite}(n_l > n_r : n = n_l + 1, n = n_r + 1)
\end{align*}

(heigh of binary tree)

bfac(x, b) := ite(x = nil, 0, \exists l, r, n_l, n_r : l = left(x) \wedge r = right(x).

\begin{align*}
\text{height}(l, n_l) \wedge \text{height}(r, n_r) \wedge b = n_r - n_l)
\end{align*}

(avl tree)

avl(x) := ite(x = nil, T, \exists l, r : l = left(x) \wedge r = right(x).

\begin{align*}
\text{avl}(l) \wedge \text{avl}(r) \wedge \text{bfac}(x) \in \{-1, 0, 1\} \wedge \\
x \notin Sp(\text{avl}(l)) \wedge x \notin Sp(\text{avl}(r)) \wedge \text{Sp(avl(l)}) \cap \text{Sp(avl(r))} = \emptyset
\end{align*}

(threaded tree)

ptree(x, n) := ite(x = nil, T, \exists l, r : l = left(x) \wedge r = right(x).

\begin{align*}
\{(r = \text{nil} \wedge \text{tnext}(x) = p) \lor (r \neq \text{nil} \wedge \text{tnext}(x) = r)\} \wedge \\
\text{ptree}(x) \wedge \text{ptree}(r, p) \wedge
\end{align*}

(auxiliary definition)

\begin{align*}
x \notin Sp(\text{ptree}(l, x)) \wedge x \notin Sp(\text{ptree}(r, p)) \wedge \text{Sp(\text{ptree}(l, x)}) \cap \text{Sp(\text{ptree}(r, p))} = \emptyset
\end{align*}

For tree

Fig. 4. Example definitions of data-structures and other predicates in Frame Logic.

tree node to the in-order successor in the tree; these pointers go from leaves to ancestors arbitrarily far away in the tree, making it a nontrivial definition.

We believe that FL formulas naturally and succinctly express these data-structures and their properties, making it an attractive logic for annotating programs.

4 PROGRAMS AND PROOFS

In this section, we develop a program logic for a while-programming language that can destructively update heaps. We assume that location variables are denoted by variables of the form x and y, whereas variables that denote other data (which would correspond to the background sorts in

\textsuperscript{6}\cup_m denotes multiset union.
\[
S ::= x := c \mid x := y \mid x := y.f \mid v := be \mid x.f := y
\mid \text{alloc}(x) \mid \text{free}(x) \mid \text{if} be \text{ then } S \text{ else } S \mid \text{while } be \text{ do } S \mid S ; S
\]

Fig. 5. Grammar of while programs. \(c\) is a constant location, \(f\) is a field pointer, and \(be\) is a background expression. In our logic, we model every field \(f\) as a function \(f()\) from locations to the appropriate sort.

our logic) are denoted by \(v\). We omit the grammar to construct background terms and formulas and simply denote such “background expressions” with \(be\) and clarify the sort when it is needed. Finally, we assume that our programs are written in Single Static Assignment (SSA) form, which means that every variable is assigned to at most once in the program text. The grammar for our programming language is in Figure 5.

4.1 Operational Semantics

In this section, we will discuss the operational semantics of our programs. First, we extend the signature of our logic by an infinite set \(\text{New} = \{\text{new} i \mid i \in \mathbb{N}\}\) of constants of the type of the foreground sort, i.e., heap locations. We also add constants \(\{\text{default} f\}_{f \in F}\) to use as “default” values of functions in \(F\).

A configuration in our operational semantics is of the form \((M, H, U)\), where \(M\) is a model that contains interpretations for the store and the heap. The store is a partial map that interprets variables, constants, and non-mutable functions over universes of the appropriate sorts. The heap is a total map on the domain of locations that interprets mutable functions. \(H\) is a subset of (the universe of) locations denoting the set of allocated locations, and \(U\) is a subset of locations denoting unallocated locations that can be allocated in the future. Last, we introduce a special configuration \(\bot\) to denote an error state.

A configuration \((M, H, U)\) is valid if:

- All program variables of the location sort (including \(\text{nil}\)) map only to locations not in \(U\).
- \(U\) does not intersect with \(H\).
- \(U\) is infinite.
- The constants \(\text{new} i\) are interpreted to distinct locations (i.e., no two constants in \(\text{New}\) are interpreted to the same location) that are neither in \(H\) nor in \(U\). We will later use these constants to design our program logic rule for allocation.
- It is not possible to reach any location in \(U\) or \(\text{New}\) from any location in \(H\). Simply, locations in \(H\) do not point to locations in \(U\) or \(\text{New}\).
- Locations in \(U\) as well as those interpreted by \(\text{New}\) have default values under functions, defined by the interpretation of the constants \(\{\text{default} f\}_{f \in F}\).

We denote this by \(\text{valid}(M, H, U)\). We will demand that initial configurations of programs are valid and maintain this by constructing our operational semantics rules to ensure that reachable configurations of programs are valid. Observe that validity of a configuration is expressible in first-order logic.

The full operational semantics are in Figure 6. We abuse notation by using \(M(x)\) to mean the value of \(x\) as interpreted by the model \(M\), as well as using the expression \(M[x \mapsto y]\) to denote that the model is updated with the variable \(x\) now storing the value stored by the variable/expression \(y\) in the original model.

\(\bot\) is a sink state, and every statement on \(\bot\) transitions to \(\bot\). The pointer lookup rule changes the store where the variable \(x\) now maps to \(f(y)\), provided \(y\) is a location that is allocated. The pointer modification rule modifies the heap on the function \(f\), where the store’s interpretation for \(x\) now maps to the store’s interpretation for \(f(y)\) (again, provided \(x\) is a location that is allocated). The allocation rule is the only nondeterministic rule in the operational semantics, as there is a transition.
for each $a \in U$. For each such $a$, the store is modified where $x$ now points to $a$. Additionally, the heap is modified for each function $f$ where the newly allocated $a$ maps to the default value under each $f$. Further, note that, since freed elements are not added back to $U$, freed locations cannot be reallocated. However, since $U$ is infinite, and dereferenced pointers must be in $H$, this does not pose a problem. All other rules are straightforward.

Note that, when side conditions are violated as in the lookup rule or pointer modification rule, the configuration transitions to $\perp$, which denotes an abort or fault configuration. These faulting transitions are crucial for the soundness of the frame rule. The allocation rule, however, will always succeed for valid configurations, because we demand that $U$ is infinite for valid configurations.

### 4.2 Triples and Validity

We express specifications of programs using triples of the form \( \{ \alpha \} S \{ \beta \} \), where $\alpha$ and $\beta$ are FL formulas in our extended signature, and $S$ is a program in our while-language. However, we restrict...
the formulas that can appear in the specifications. First, we disallow atomic relations between locations. Second, we disallow functions from a background sort to the foreground sort (see Section 3). Finally, quantified formulas can have supports as large as the entire heap, but we want our program logic to cover a more practical fragment without compromising expressive power. Thus, we require guards in quantification to be of the form \( f(z') = z \), where \( z \) is the quantified variable. We will maintain this as an invariant in the formulas that we generate for, say, weakest preconditions. Apart from these restrictions, we will also assume that formulas only feature unary functions for ease of presentation.

We define a triple to be valid if every valid configuration with heaplet being precisely the support of \( \alpha \), when acted on by the program, yields a configuration with heaplet being the support of \( \beta \). More formally, a triple is valid if for every valid configuration \((M, H, U)\) such that \( M \models \alpha \), \( H = [Sp(\alpha)]_M \):

1. it is never the case that the abort state \( \bot \) is encountered in the execution on \( S \);
2. if \((M, H, U)\) transitions to \((M', H', U')\) on \( S \), then \( M' \models \beta \) and \( H' = [Sp(\beta)]_{M'} \).

Note that the post-configuration \((M', H', U')\) will also be valid as a consequence of our transition rules, using the fact that \((M, H, U)\) is valid.

4.3 Program Logic

First, we define a set of local rules and rules for conditionals, while, sequence, consequence, and framing. Recall we assume there is at least one mutable function \( f \in F_m \) (see Section 3) in our signature.

**Assignment:** \( \{\top\} \ x := \ y \ {\{x = y\}} \ \{\top\} \ x := \ c \ {\{x = c\}} \)

**Lookup:** \( \{f(y) = f(y)\} \ x := \ y . f \ {\{x = f(y)\}} \)

**Mutation:** \( \{f(x) = f(x)\} \ x . f := \ y \ {\{f(x) = y\}} \)

**Allocation:** \( \{\top\} \ alloc(x) \ \{\bigwedge_{f \in F} f(x) = \text{default}\} \)

**Deallocation:** \( \{f(x) = f(x)\} \ free(x) \ {\top}\)

**Conditional:** \( \{\alpha\} \text{ if be then } S \text{ else } T \ {\{\beta\}} \)

\[\alpha \land \text{be} \ \{\alpha\} \)

**While:** \( \{\alpha\} \text{ while be do } S \ {\{\neg \text{be} \land \alpha\}} \)

\[\alpha \ \{\beta\} \ \{\beta\} \ T \ {\mu} \)

**Sequence:** \( \{\alpha\} \ S ; \ T \ {\mu} \)

\[\alpha' \implies \alpha \ \text{Sp}(\alpha) = \text{Sp}(\alpha') \]

\[\beta \implies \beta' \ \{\alpha\} \ S \ {\beta} \ \text{Sp}(\beta) = \text{Sp}(\beta') \)

**Consequence:** \( \{\alpha'\} \ S \ {\beta'} \)

\[\text{Sp}(\alpha) \cap \text{Sp}(\mu) = \emptyset \ \{\alpha\} \ S \ {\beta} \ \text{vars}(S) \cap \text{fv}(\mu) = \emptyset \)

**Frame:** \( \{\alpha \land \mu\} \ S \ {\beta \land \mu} \)

The above rules are intuitively clear and are similar to the local rules in separation logic [63]. The rules for statements capture their semantics using minimal/tight heaplets, and the frame rule allows proving triples with larger heaplets. Some seemingly trivial preconditions as in the lookup, mutation, and allocation rules are added to ensure tight heaplets, i.e., that the support of the
precondition is equal to the support of the postcondition, modulo any alloc/free statements. In
the rule for alloc, the postcondition says that the newly allocated location has default values for
all pointer fields and data fields (denoted as \(\text{def}_f\)). The soundness of the frame rule relies crucially
on the frame theorem for FL (Theorem 3.7). The full soundness proof can be found in Section 4.7.

**Theorem 4.1.** The above rules are sound with respect to the operational semantics.

### 4.4 Weakest-precondition Proof Rules

We now turn to the much more complex problem of designing rules that give weakest preconditions for arbitrary postconditions, for loop-free programs. In separation logic, such rules resort to
using the magic wand operator \(\vdash\) [21, 51, 52, 63]. The magic wand is a complex operator whose
semantics calls for second-order quantification [14] over arbitrarily large submodels. In our setting,
our main goal is to show that FL is itself capable of expressing weakest preconditions of postcondi-
tions written in FL.

First, we define a notion of **Weakest Tightest Precondition (WTP)** of a formula \(\beta\) with respect
to each command that can figure in a basic block: assignment, lookup, mutation, allocation, and
deallocation. To define this notion, we first define the notion of a preconfiguration:

**Definition 4.2.** The preconfigurations corresponding to a valid configuration \((M, H, U)\) with re-
spect to a program \(S\) are a set of valid configurations of the form \((M_p, H_p, U_p)\) such that when \(S\)
is executed on \(M_p\) with unallocated set \(U_p\) it dereferences only locations in \(H_p\) and results (using
the operational semantics rules) in \((M, H, U)\). That is:

\[
\text{preconfigurations}((M, H, U), S) = \{(M_p, H_p, U_p) \mid \text{valid}(M_p, H_p, U_p) \text{ and } (M_p, H_p, U_p) \implies (M, H, U))\}.
\]

**Definition 4.3.** We say that \(\alpha\) is a **Weakest Tightest Precondition (WTP)** of a formula \(\beta\) with
respect to a program \(S\) if the set of all valid configurations that satisfy \(\alpha\) is the same as the set of all
preconfigurations of all valid configurations that satisfy \(\beta\), with the addition of similar conditions
on the allocated locations. More formally:

\[
\{(M_p, H_p, U_p) \mid M_p \models \alpha, H_p = [\text{Sp}(\alpha)]_{M_p}, \text{valid}(M_p, H_p, U_p)\} = \{C \mid C \in \text{preconfigurations}(C_{\text{post}}, S) \text{ for some } C_{\text{post}} = (M, H, U), M \models \beta, H = [\text{Sp}(\beta)]_{M}, \text{valid}(M, H, U)\}.
\]

With the notion of weakest tightest preconditions, we define global program logic rules for each
command of our language. In contrast to local rules, global specifications contain heaplets that may
be larger than the smallest heap on which one can execute the command.

Intuitively, a WTP of \(\beta\) for lookup states that \(\beta\) must hold in the precondition when \(x\) is inter-
preted as \(x'\), where \(x' = f(y)\), and further that the location \(y\) must belong to the support of \(\beta\).
The rules for mutation and allocation are more complex. For mutation, we define a transformation
\(MW^{x\cdot f\neg y}(\beta)\) that evaluates a formula \(\beta\) in the pre-state as though it were evaluated in the post-
state. We similarly define such a transformation \(MW^{\text{alloc}(x)}\) for allocation. We will define these in
detail later. Finally, the deallocation rule ensures \(x\) is not in the support of the postcondition. The
conjunct \(f(x) = f(x)\) is provided to satisfy the tightness condition, ensuring that the support of
the precondition is the support of the postcondition with the addition of \(\{x\}\). The rules can be seen
below, and the proof of soundness for these global rules can be found in Section 4.7.

**Assignment-G:** \(\beta[y/x] \ x := y \ \{\beta\} \quad \beta[c/x] \ x := c \ \{\beta\}\)

**Lookup-G:** \(\exists x' : x' = f(y). (\beta \land y \in \text{Sp}(\beta))[x'/x] \ x := y.f \ \{\beta\}\)

(where \(x'\) does not occur in \(\beta\))
The rules above suffixed with -G are sound w.r.t. the operational semantics. And, each precondition corresponds to the weakest tightest precondition of \( \beta \).

### 4.5 Definitions of MW Primitives: Mutation

Recall that the MW\(^7\) primitives \( MW^{x,f:=y} \) and \( MW^{\text{alloc}(x)}_v \) need to evaluate a formula \( \beta \) in the pre-state, as it would evaluate in the post-state after mutation and allocation statements. The definition of \( MW^{x,f:=y} \) is as follows:

\[
MW^{x,f:=y}(\beta) = \beta[\lambda z. \text{ite}(z = x : \text{ite}(f(x) = f(x) : y, y), f(z))]/f].
\]

The \( \beta[\lambda z.p(z)/f] \) notation is shorthand for saying that each occurrence of a term of the form \( f(t) \), where \( t \) is a term, is substituted (recursively, from inside out) by the term \( p(t) \). The precondition essentially evaluates \( \beta \) taking into account \( f \)'s transformation, but we use the \( \text{ite} \) expression with a tautological guard \( f(x) = f(x) \) (which has the support containing the singleton \( x \)) to preserve the support (see Section 4.7, Lemma 4.7). The definition of \( MW^{\text{alloc}(x)}_v \) is similar, but involves a few subtleties.

### 4.6 Definitions of MW Primitives: Allocation

We have already seen the definition of \( MW^{x,f:=y} \) in Section 4.5. Observe that the inner guard contains the expression \( f(x) = f(x) \), which is a tautology that only serves to include \( x \) in the support of the transformed formula (as is required by the weakest tightest precondition definition). This is similar to the Separation Logic syntax \( x \mapsto _{-} \). Of course, one can use syntax sugars such as \( \text{acc}(x) \) available in tools like Viper [47] to make the intent clearer.

We will detail the construction of \( MW^{\text{alloc}(x)}_v \) in this section.

\( MW^{\text{alloc}(x)}_v \), like \( MW^{x,f:=y} \), is also meant to evaluate a formula in the pre-state as though it were evaluated in the post-state. However, note that the support of this formula must not contain the allocated location (say, \( x \)). Since we know from the operational semantics of allocation that the allocated location is going to point to default values, we can proceed similarly as we did for the previous definition, identify terms evaluating to \( f(x) \) and replace them with the default value (under \( f \)). This has the intended effect of evaluating to the same value as in the post-state while removing \( x \) from the support.

However, this approach fails when we apply it to support expressions (since removing \( x \) from the support guarantees that we would no longer compute the “same” value as of that in the post-state). In particular, a subformula of the form \( t \in \text{Sp}(\gamma) \) may be falsified by that transformation. To handle this, we identify when \( x \) might be in the support of a given expression and replace it with \( v \) (which is given as a parameter) such that neither \( x \) nor \( v \) is dereferenced and will not be in the support of the resulting transformation. We do this syntactic replacement of \( x \) with \( v \) for formulas not within support expressions as well. This has the effect of eliminating \( x \) altogether from the formula.

We define \( MW^{\text{alloc}(x)}_v \) inductively. We first consider the case where \( \beta \) does not contain any subformulas involving support expressions or inductive definitions. Then, we have \( MW^{\text{alloc}(x)}_v \) defined

---

\(^7\)The acronym MW is a shout-out to the Magic Wand operator, as these serve a similar function, except that they are definable in FL itself.
\[ \begin{align*}
SP_{x^0}(\gamma, z) & := \bot \quad \text{for a constant } c \\
SP_{v^0}(\gamma, z) & := \bot \quad \text{for a variable } w \\
SP_{f(t)}(\gamma, z) & := \left\{ \begin{array}{ll}
(z = MW^\text{alloc}(x)(t)) \lor SP^x_{t^1}(\gamma, z) \land \left( MW^\text{alloc}(x)(f(t) = f(t)) \right) & \text{if } f \in F_m \\
SP^x_{t^2}(\gamma, z) & \text{if } f \notin F_m
\end{array} \right.
\end{align*} \]

\[ SP^x_{\beta}(\gamma, z) := SP^x_{\beta}(\gamma, z) \]

\[ SP^x_{(t_1 = t_2)}(\gamma, z) := SP^x_{t_1}(\gamma, z) \lor SP^x_{t_2}(\gamma, z) \]

\[ SP^x_{(\gamma)}(\gamma, z) := \bot \]

\[ SP^x_{(\bot)}(\gamma, z) := \bot \]

\[ SP^x_{R(\bar{w})}(\gamma, z) := SP^x_{PR(\bar{w})}(\gamma, z) \lor \bigwedge_{i=1}^n SP^x_{t_i}(\gamma, z) \quad \text{for } R \in I \text{ with definition } R(\bar{w}) := \rho_R(\bar{w}) \]

\[ SP^x_{\beta \land \beta}(\gamma, z) := SP^x_{\beta}(\gamma, z) \lor SP^x_{\beta}(\gamma, z) \]

\[ SP^x_{\beta \lor \beta}(\gamma, z) := SP^x_{\beta}(\gamma, z) \lor SP^x_{\beta}(\gamma, z) \]

\[ SP^x_{\text{ite}(y; \beta_1, \beta_2)}(\gamma, z) := SP^x_Y(\gamma, z) \lor \text{ite}(MW^\text{alloc}(x)(\gamma) : SP^x_{\beta_1}(\gamma, z), SP^x_{\beta_2}(\gamma, z)) \]

\[ SP^x_{\text{ite}(y; t_1, t_2)}(\gamma, z) := SP^x_Y(\gamma, z) \lor \text{ite}(MW^\text{alloc}(x)(\gamma) : SP^x_{t_1}(\gamma, z), SP^x_{t_2}(\gamma, z)) \]

\[ SP^x_{\exists y : \mathcal{P}(\gamma, z)} := \exists w : \left( MW^\text{alloc}(x)(\gamma) \land SP^x_{w}(\gamma, z) \right) \]

Fig. 7. Definition of \( SP^x_{v^0} \) for use in \( MW^\text{alloc}(x) \).

as follows:

\[ MW^\text{alloc}(x)(\beta) = \beta[v/x][\lambda z. \ \text{ite}(z = v : \text{default}, f(z))/f]_{f \in F} \]

where this means for each instance of a (mutable or immutable) function \( f \) in \( \beta \), we replace \( f(x) \) with a default value. We also replace all free instances of \( x \) in \( \beta \) with \( v \).

If \( Sp(\gamma) \) is a subterm of \( \beta \), then we translate it to a formula \( SP^x_{\gamma} \) inductively as in Figure 7. This definition is very similar to the translation of FL formulas to FO-RD in Figure 3, where we replace free instances of \( x \) with \( v \). Since this is a relation, we must transform membership to evaluation, i.e., transform expressions of the form \( t \in Sp(\gamma) \) to \( SP^x_{\gamma}(\gamma, MW^\text{alloc}(x)(t)) \) where \( \gamma \) are the free variables (we transform inductively—at the highest level free variables will be program/ghost variables). We also transform union of support expressions to disjunction of the corresponding relations, equality to (quantified) double implication, and so on.

For a subterm of \( \beta \) of the form \( I(\bar{T}) \) where \( I \) is an inductive definition with body \( \rho_I \), we translate it to \( I'(MW^\text{alloc}(x)(\bar{T})) \) where the body of \( I' \) is defined as \( MW^\text{alloc}(x)(\rho_I) \).

The above cases can be combined with Boolean operators and if-then-else, which \( MW^\text{alloc}(x) \) distributes over.

Last, to design the program logic rule, we have to decide the value of the parameter \( v \). The idea is that \( v \) (which is essentially \( x \)) will hold the location that is going to be allocated. Since we do not know which one of the locations in the unallocated set will be actually allocated next, and \( v \) will not feature in the support of the \( MW^\text{alloc}(x) \) formula by construction, it is enough to choose an element that simply mirrors the behavior of the location to be allocated, in that it will evaluate the same way under any function and is different from any location that has ever been allocated. This is where we use our “dummy” location constants \( new_i \) with which we extended our signature (see Section 4.1). We also maintain the invariant that these dummy locations will not feature in the
support of any preconditions that we generate. Note that we ensure that these dummy locations are indeed different by first demanding that the constants bear distinct values (see Section 4.1) and ensuring that we use a constant that does not appear in the given postcondition (see Section 4.4).

4.7 Program Logic Proofs

In this section, we present soundness proofs for the global rules developed in Section 4.4. We often drop $U$ when referring to a configuration $(M, H, U)$ for ease of presentation, since $U$ is only modified by the allocation rule.

**Theorem 4.5 (Lookup Soundness).** Let $M$ be a model and $H$ a sub-universe of locations such that

$$M \models \exists x' : x' = f(y). (\beta \land y \in Sp(\beta))[x'/x]$$

$$H = [Sp(\exists x' : x' = f(y)). (\beta \land y \in Sp(\beta))[x'/x]]_M.$$

Then, $(M, H) \xrightarrow{x=y,f} (M', H')$, $M' \models \beta$, and $H' = [Sp(\beta)]_{M'}$.

**Proof.** Observe that $[y]_M \in H$, since $y$ is in the support of the precondition. Therefore, we know $(M, H) \xrightarrow{x=y,f} (M', H')$ where $M' = M[x \mapsto [f(y)]_M]$ and $H' = H$. Next, note that if there is a formula $\alpha$ (or term $t$) where $x$ is not a free variable of $\alpha$ (or $t$), then $M$ and $M'$ have the same valuation of $\alpha$ (or $t$). This is true, because the semantics of lookup only changes the valuation for $x$ on $M$. In particular, $M' \models \exists x' : x' = f(y). (\beta \land y \in Sp(\beta))[x'/x]$. Thus,

$$M' \models \exists x' : x' = f(y). (\beta \land y \in Sp(\beta))[x'/x]$$

$$\implies M'[x' \mapsto c]$$

$$\models x' = f(y) \land (\beta \land y \in Sp(\beta))[x'/x]$$

(for some $c$)

$$\implies M'[x' \mapsto [f(y)]_{M'}] \models (\beta \land y \in Sp(\beta))[x'/x]$$

(since $f$ is a function)

$$\implies M'[x' \mapsto [x]_{M'}] \models (\beta \land y \in Sp(\beta))[x'/x]$$

(operational semantics)

$$\implies M'[x' \mapsto [x]_{M'}] \models \beta \land y \in Sp(\beta)$$

($\beta$ does not mention $x'$)

$$\implies M' \models \beta.$$

The heaplet condition follows from a similar argument. Specifically,

$$H' = H$$

$$= [Sp(\exists x' : x' = f(y). (\beta \land y \in Sp(\beta))[x'/x])_M$$

$$= [Sp(\exists x' : x' = f(y). (\beta \land y \in Sp(\beta))[x'/x]]_{M'}$$

(does not mention $x$)

$$= ([y]_{M'}) \cup [Sp((\beta \land y \in Sp(\beta))[x'/x])][x' \mapsto [f(y)]_{M'}]$$

(def of $Sp$)

$$= ([y]_{M'}) \cup [Sp((\beta \land y \in Sp(\beta))[x'/x])][x' \mapsto [x]_{M'}]$$

(operational semantics)

$$= ([y]_{M'}) \cup [Sp(\beta \land y \in Sp(\beta))[x'/x]]_{M'}$$

(similar reasoning as above)

$$= [Sp(\beta)]_{M'}.$$  

(since $M' \models y \in Sp(\beta)$ from above)

$\square$
Theorem 4.6 (WTP Lookup). Let $M, M'$ be models with $H, H'$ sub-universes of locations (respectively) such that $(M, H) \xrightarrow{\beta} (M', H')$, $M \models \beta$ and $H' = \llbracket \text{Sp}(\beta) \rrbracket_{M'}$. Then

$$M \models \exists x': x' = f(y). (\beta \land y \in \text{Sp}(\beta))[x'/x]$$

(weakest-pre)

$$H = \llbracket \text{Sp}(\exists x': x' = f(y). (\beta \land y \in \text{Sp}(\beta))) \rrbracket_M.$$  

(tightest-pre)

Proof. Both parts follow by simply retracing steps in the above proof. The weakness claim follows from the first part of the proof above, where all implications can be made bidirectional (using operational semantics rules, definition of existential quantifier, etc.). The tightness claim follows immediately from the second part of the proof above as all steps involve equalities. $\Box$

For soundness of the pointer modification rules, we prove the following lemma:

Lemma 4.7. Given a formula $\beta$ (term $t$) and configurations $(M, H)$ and $(M', H')$ such that $(M, H)$ transforms to $(M', H')$ on the command $x.f := y$, then $\llbracket MW^{x.f := y}(\beta) \rrbracket_M = \llbracket \beta \rrbracket_{M'}$. Additionally, $\llbracket \text{Sp}(MW^{x.f := y}(\beta)) \rrbracket_M = \llbracket \text{Sp}(\beta) \rrbracket_{M'}$. Both equalities hold for terms $t$ as well.

Proof. Induction on the structure of $\beta$, unfolding $MW^{x.f := y}(\beta)$ accordingly. We discuss one interesting case here, namely, when $\beta$ has a subterm of the form $f(t)$. Now, we have two cases, depending on whether $\llbracket MW^{x.f := y}(t) \rrbracket_{M}(t) = \llbracket x \rrbracket_{M}$. If it does, then

$$\llbracket MW^{x.f := y}(f(t)) \rrbracket_M = \llbracket \text{ite}(MW^{x.f := y}(t)) : \text{ite}(f(x) = f(x) : y, y), f(MW^{x.f := y}(t))) \rrbracket_M$$

(definition)

$$= \llbracket \text{ite}(f(x) = f(x) : y, y) \rrbracket_M$$

(assumption)

$$= \llbracket y \rrbracket_M$$

$$= \llbracket y \rrbracket_{M'}$$

$$= \llbracket f \rrbracket_{M'}(\llbracket x \rrbracket_{M'})$$

(def of $f$ on $M'$)

$$= \llbracket f \rrbracket_{M'}(\llbracket t \rrbracket_{M'})$$

(induction hypothesis)

$$= \llbracket f(t) \rrbracket_{M'}.$$  

The proof for the cases when $\llbracket MW^{x.f := y}(t) \rrbracket_{M}(t) \neq \llbracket x \rrbracket_{M}$ and the heaplet equality claims are similar, and all other cases are trivial. $\Box$

Theorem 4.8 (Mutation Soundness). Let $M$ be a model and $H$ a sub-universe of locations such that

$$M \models MW^{x.f := y}(\beta \land x \in \text{Sp}(\beta))$$

$$H = \llbracket \text{Sp}(MW^{x.f := y}(\beta \land x \in \text{Sp}(\beta))) \rrbracket_M.$$

Then, $(M, H) \xrightarrow{\beta} (M', H')$, $M \models \beta$, and $H' = \llbracket \text{Sp}(\beta) \rrbracket_{M'}$.

Proof. From the definition of the transformation $MW^{x.f := y}$, we have that $MW^{x.f := y}(\beta \land x \in \text{Sp}(\beta))$ will be transformed to the same formula as $MW^{x.f := y}(\beta \land x \in \text{Sp}(MW^{x.f := y}(\beta)))$, the heaplet of which, since the formula holds on $M$, contains $x$. Therefore, $x \in H$ and from the operational semantics, we have that $(M, H) \xrightarrow{\beta} (M', H')$ for some $(M', H')$ such that $H = H'$. 

ACM Transactions on Programming Languages and Systems, Vol. 45, No. 2, Article 7. Publication date: May 2023.
From Lemma 4.7, we have that $M' \models \beta \land x \in \text{Sp}(\beta)$, since $M$ models the same. In particular $M' \models \beta$. Moreover, we have

$$H' = H$$

(operational semantics)

$$= \left[ \text{Sp}(M \cdot W^{x \cdot f := y}(\beta \land x \in \text{Sp}(\beta))) \right]_M$$

(given)

$$= \left[ \text{Sp}(\beta \land x \in \text{Sp}(\beta)) \right]_{M'}$$

(Lemma 4.7)

$$= \left[ \text{Sp}(\beta) \right]_{M'}.$$  

(-semantics of H operator)

Therefore, $M' \models \beta$ and $H' = \left[ \text{Sp}(\beta) \right]_{M'}$, which makes our pointer mutation rule sound. □

**Theorem 4.9 (WTP Mutation).** Let $M, M'$ be models with $H, H'$ sub-universes of locations (respectively) such that $(M, H) \xrightarrow{x \cdot f := y} (M', H')$, $M' \models \beta$ and $H' = \left[ \text{Sp}(\beta) \right]_{M'}$. Then

$$M \models M \cdot W^{x \cdot f := y}(\beta \land x \in \text{Sp}(\beta))$$

(weakest-pre)

$$H = \left[ \text{Sp}(M \cdot W^{x \cdot f := y}(\beta \land x \in \text{Sp}(\beta))) \right]_M.$$  

(tightest-pre)

**Proof.** From the operational semantics, we have that $(M, H) \xrightarrow{x \cdot f := y} (M', H')$ only if $x \in H$ and $H = H'$. Therefore, $x \in H' = \left[ \text{Sp}(\beta) \right]_{M'}$ (given), which in turn implies that $M' \models \beta \land x \in \text{Sp}(\beta)$ as well as $H' = \left[ \text{Sp}(\beta \land x \in \text{Sp}(\beta)) \right]_{M'}$. Applying Lemma 4.7 yields the result. □

**Lemma 4.10.** Given a formula $\beta$ (or term $t$) and configurations $(M, H, U)$ and $(M', H', U')$ such that $(M, H, U) \xrightarrow{\text{alloc}(x)} (M', H', U')$, it is the case that \( \left[ \text{Sp}^{x \cdot \nu}(\bar{\nu}, M \cdot W^{\text{alloc}(x)}_{U}(t)) \right]_{M[U \mapsto a]} \) if \( \left[ t \in \text{Sp}(\beta) \right]_{M'} \), where $a = \left[ x \right]_{M'}$ and $\bar{\nu}$ are the free variables in $M \cdot W^{\text{alloc}(x)}_{U}(\beta)$. Additionally, \( \left[ \text{Sp}(\beta) \right]_{M[U \mapsto a]} = \left[ \text{Sp}(\beta \setminus \{x\}) \right]_{M'} \), where $z$ is a free variable. Both equalities hold for terms $t$ as well.

**Proof.** Induction on the structure of $\beta$ and using the construction in Figure 7. For the second claim about the support of $Sp^{x \cdot \nu}$, the fact that we only allow specific kinds of guards is crucial in the inductive case of the existential quantifier. □

**Lemma 4.11.** Given configurations $(M, H, U), (M', H', U')$ such that $(M, H, U) \xrightarrow{\text{alloc}(x)} (M', H', U')$ and a formula $\beta$ (or term $t$), we have \( \left[ M \cdot W^{\text{alloc}(x)}_{U}(\beta) \right]_{M[U \mapsto a]} = \left[ \beta \right]_{M'} \), where $a = \left[ x \right]_{M'}$. Additionally, \( \left[ \text{Sp}(M \cdot W^{\text{alloc}(x)}_{U}(\beta)) \right]_{M[U \mapsto a]} = \left[ \text{Sp}(\beta) \setminus \{x\} \right]_{M'} \). Both equalities hold for terms $t$ as well.

**Proof.** First, we split on the structure of $\beta$, as the definition of $M \cdot W^{\text{alloc}(x)}_{U}$ differs depending on the form of $\beta$. For subformulas with no support expressions or inductive definitions, the proof follows from the syntactic definition of $M \cdot W^{\text{alloc}(x)}_{U}$ and is very similar to Lemma 4.7. It is important to note that $a$ is a location different from any of the locations ever allocated and is therefore different from any location held in any program variable or reachable by any recursive definition. This is crucial in the case of handling the atomic equality/disequality formulas.

Subformulas with support expressions follow by construction using Lemma 4.10, and formulas with inductive definitions follow by construction as well. Boolean combinations and if-then-else follow using the inductive hypothesis. □

We are now ready to prove the soundness and WTP property of the allocation rule. This will be slightly different from the other soundness theorems, because it reasons only about configurations reachable by a program or a valid initial state. This strengthening of the premise is not an issue, since we will only ever execute commands on such states. We shall first prove a lemma.
Lemma 4.12. Let \( \beta \) be any formula within our restricted fragment (Section 4.2) and \((M, H, U)\) be a valid configuration. Then, for any locations \( a_1, a_2 \in U \cup \text{New}:\)

\[
\begin{align*}
\left[ MW^{\text{alloc}(x)}_v(\beta) \right]_{M[v \mapsto a_1]} &= \left[ MW^{\text{alloc}(x)}_v(\beta) \right]_{M[v \mapsto a_2]} \quad \text{and} \\
\left[ Sp(MW^{\text{alloc}(x)}_v(\beta)) \right]_{M[v \mapsto a_1]} &= \left[ Sp(MW^{\text{alloc}(x)}_v(\beta)) \right]_{M[v \mapsto a_2]}.
\end{align*}
\]

Proof. The proof follows by a simple inductive argument on the structure of \( \beta \). First, observe that in any model if \( v \) is interpreted to a hitherto unallocated location (either from \( U \) or one of the dummy constants), then it is never contained in \( MW^{\text{alloc}(x)}_v(\beta) \), since it is never dereferenced. Therefore, all we are left to prove is that the actual value of \( v \) (between choices in \( U \cup \text{New} \)) influences neither the truth value nor the support of the formula. The key case is that of \texttt{ite} expressions where the value of \( v \) can influence the truth of the guard. This case can be resolved using the observation that, since \((M, H, U)\) is a valid configuration, the value of any unallocated location can never equal that of a program variable. Since we have no atomic relations either in our restricted fragment, any two values in \( U \cup \text{New} \) are indistinguishable by a formula in this fragment.

In particular, any \texttt{ite} expressions that depend on the value of \( v \) either compare it with a term over a program variable—which is never equal—or compare it with a quantified variable—which itself only takes on values allowed by the guard of the quantification that, inductively, does not distinguish between values in \( U \cup \text{New} \). \( \square \)

Theorem 4.13 (Allocation Soundness). Let \((M, H, U)\) be a valid configuration such that

\[
M \models MW^{\text{alloc}(x)}_{\text{new}}(\beta),
\]

\[
H = \left[ Sp(MW^{\text{alloc}(x)}_{\text{new}}(\beta)) \right]_M,
\]

such that \texttt{new} does not appear in \( \beta \). Then, \((M, H, U) \xrightarrow{MW^{\text{alloc}(x)}_{\text{new}}} (M', H', U \setminus \{x\}_{M'})\), \( M' \models \beta \) and \( H' = \left[ Sp(\beta) \right]_{M'} \).

Proof. It is easy to see that \((M, H, U) \xrightarrow{MW^{\text{alloc}(x)}_{\text{new}}} (M', H', U \setminus \{x\}_{M'})\) for some \( M', H' \), since \( U \) is infinite for any valid configuration.

Let \( a \) be the actual location allocated, i.e., \( a = \{x\}_{M'} \). Clearly, \( a \in U \) by the operational semantics. Then, we have:

\[
M \models MW^{\text{alloc}(x)}_{\text{new}}(\beta) \implies M[v \mapsto \text{new}] \models MW^{\text{alloc}(x)}_v(\beta) \implies M[v \mapsto a] \models MW^{\text{alloc}(x)}_v(\beta) \quad \text{(Lemma 4.12)}
\]

\[
M' \models \beta. \quad \text{(Lemma 4.11)}
\]

For the support claim, we have:

\[
H = \left[ Sp(MW^{\text{alloc}(x)}_{\text{new}}(\beta)) \right]_M = \left[ Sp(MW^{\text{alloc}(x)}_v(\beta)) \right]_{M[v \mapsto \text{new}]} = \left[ Sp(MW^{\text{alloc}(x)}_v(\beta)) \right]_{M[v \mapsto a]} = \left[ Sp(\beta) \setminus \{x\} \right]_{M'}. \quad \text{(Lemma 4.12)}
\]
Now, $H' = H \cup \{[x]_{M'}\}$ (by operational semantics) = $[Sp(\beta) \setminus \{x\}]_{M'} \cup ([x]_{M'}) = [Sp(\beta)]_{M'}$, as desired.

**Theorem 4.14 (WTP Allocation).** Let $(M, H, U)$ and $(M', H', U \setminus [x]_{M'})$ be valid configurations such that

$$\begin{array}{c}
(M, H, U) \xrightarrow{\text{alloc}(x)} (M', H', U \setminus [x]_{M'}) \\
M' \models \beta \text{ and } H' = [Sp(\beta)]_{M'}.
\end{array}$$

Then,

$$\begin{array}{c}
M \models MW_{\text{new}}^{\text{alloc}(x)}(\beta) \\
H = [Sp(MW_{\text{new}}^{\text{alloc}(x)}(\beta))]_M
\end{array}$$

for some $j$ such that $\text{new}_j$ does not appear in $\beta$.

**Proof.** The first claim follows easily from an application of Lemma 4.11 followed by an application of Lemma 4.12. For the second claim, observe that as done in the proof above for Theorem 4.13, we can prove that $[Sp(\beta) \setminus \{x\}]_{M'} = [Sp(MW_{\text{new}}^{\text{alloc}(x)}(\beta))]_M$. The proof concludes by observing that by the operational semantics, we have $H = H' \setminus ([x]_{M'}) = [Sp(\beta)]_{M'} \setminus ([x]_{M'}) = [Sp(\beta) \setminus \{x\}]_{M'}$. □

**Theorem 4.15 (Deallocation Soundness).** Let $M$ be a model and $H$ a sub-universe of locations such that

$$\begin{array}{c}
M \models \beta \land x \notin Sp(\beta) \land f(x) = f(x) \\
H = [Sp(\beta) \land x \neq Sp(\beta) \land f(x) = f(x))]_M.
\end{array}$$

Then, $(M, H) \xrightarrow{\text{free}(x)} (M', H')$, $M' \models \beta$, and $H' = [Sp(\beta)]_{M'}$.

**Proof.** Observe that $x \in Sp(\beta) \land x \notin Sp(\beta) \land f(x) = f(x)$, i.e., $[x]_M \in H$. Therefore, we have from the operational semantics that $(M, H) \xrightarrow{\text{free}(x)} (M', H')$ such that $M' = M$ and $H' = H' \setminus ([x]_M)$. Since $M \models \beta \land x \notin Sp(\beta) \land f(x) = f(x)$, we know $M \models \beta$, which implies $M' \models \beta$. Similarly, we have:

$$\begin{array}{c}
H' = H \setminus ([x]_M) \quad \text{(operational semantics)} \\
= [Sp(\beta) \land x \neq Sp(\beta) \land f(x) = f(x))]_M \setminus ([x]_M) \\
= [Sp(\beta)]_M \cup ([x]_M) \setminus ([x]_M) \quad \text{(def of Sp)} \\
= [Sp(\beta)]_M \\
= [Sp(\beta)]_{M'} \quad \text{(M = M')}.
\end{array}$$

**Theorem 4.16 (WTP Deallocation).** Let $M, M'$ be models with $H, H'$ sub-universes of locations (respectively) such that $(M, H) \xrightarrow{\text{free}(x)} (M', H')$, $M' \models \beta$ and $H' = [Sp(\beta)]_{M'}$. Then

$$\begin{array}{c}
M \models \beta \land x \notin Sp(\beta) \land f(x) = f(x) \quad \text{(weakest-pre)} \\
H = [Sp(\beta) \land x \neq Sp(\beta) \land f(x) = f(x))]_M. \quad \text{(tightest-pre)}
\end{array}$$

**Proof.** For the first part, note that the operational semantics ensures $[x]_{M'} \notin H' = [Sp(\beta)]_{M'}$ and $M = M'$. So, $M' \models x \notin Sp(\beta)$, which implies $M \models x \notin Sp(\beta)$. Similarly, $M \models \beta$, and
\[ M \models f(x) = f(x) \], as it is a tautology. Tightness follows from similar arguments as in Theorem 4.15, again noting that \( H' = H \setminus \{ [x]_M \} \), as per the operational semantics.

**Theorem 4.1.** The four local rules (for assignment, lookup, mutation, allocation, and deallocation) given in Section 4 are sound given the global rules.

**Proof.** The validity of assignment follows immediately setting \( \beta \) to be \( x = y \) (or \( x = c \)). Instantiating with this and the precondition becomes \( y = y \), which is equivalent to \( \text{true} \) (the heaplet of both is empty).

The validity of the next (lookup) follows, since
\[
\begin{align*}
\text{wtp}(x = f(y), x := y.f) &= \exists x' : x' = f(y). (x = f(y) \land y \in Sp(x = f(y)))[x'/x] \\
&= \exists x' : x' = f(y), x' = f(y) \land y \in Sp(x' = f(y)) \\
&= f(y) = f(y) \land y \in Sp(f(y) = f(y)).
\end{align*}
\]
This is a tautology, so it is clearly implied by any precondition, in particular, the precondition \( f(y) = f(y) \). Similarly, the support of the resulting formula is the singleton \( \{y\} \), which is also the support of \( f(y) = f(y) \), as needed.

For the second local rule (mutation), we first notice that
\[
MW^{x.f := y}(f(x) = y) = \frac{ite(z = x : ite(f(x) = f(x) : y, y), f(z))/f(x)}{f(x)}
\]
Then,
\[
\text{wtp}(f(x) = y, x.f := y) = ite(x = x : ite(f(x) = f(x) : y, y), f(x)) = y \\
\land x \in Sp(ite(x = x : ite(f(x) = f(x) : y, y), f(x)) = y).
\]
The first conjunct is clearly true, since it is equivalent to \( y = y \). The second conjunct is also true, because \( Sp(ite(x = x : ite(f(x) = f(x) : y, y), f(x)) = y) = \{x\} \). Thus, this formula is also a tautology, and it is implied by the precondition \( f(x) = f(x) \). Additionally, the support of the resulting formula and the support of \( f(x) = f(x) \) is \( \{x\} \), as needed.

For the next local rule (allocation), observe that the postcondition does not have any support expressions or inductive definitions. Therefore, we have that:
\[ MW^\text{alloc}(x)(f(x) = \text{default}_f) = ite(x = x : \text{default}_f, f(x)) = \text{default}_f. \]
Observe that the support of the above expression is \( \emptyset \). The support of a conjunction of such expressions is also \( \emptyset \). This and the fact that \( MW^\text{alloc}(x) \) distributes over \( \land \) gives us:
\[
\text{wtp}\left( \bigwedge_{f \in F} f(x) = \text{default}_f \right), x := \text{alloc}()
\]
\[
= \forall v : v \notin \emptyset \implies MW^\text{alloc}(x)\left( \bigwedge_{f \in F} f(x) = \text{default}_f \right)
\]
\[
= \forall v : \bigwedge_{f \in F} ite(x = x : \text{default}_f, f(x)) = \text{default}_f,
\]
which is a tautology (as it is equivalent to \( \text{default}_f = \text{default}_f \)) and its support is \( \emptyset \), as desired.

Finally the last local rule (deallocation) follows directly from the global rule for deallocation by setting \( \beta = \top \).
Theorem 4.17 (Conditional, While Soundness). We refer the reader to a classical proof of the soundness of these rules, as in Reference [2].

Theorem 4.18 (Sequence Soundness). The Sequence rule is sound.

The proof of this theorem follows directly from the operational semantics.

Theorem 4.19 (Consequence Soundness). The Consequence rule is sound.

Proof. First, note if we cannot execute $S$, then the triple is vacuously valid. Next, assume $M \models \alpha'$. Then, because $\alpha' \implies \alpha$, we know $M \models \alpha$. So, if we execute $S$ and result in $M'$, then we know $M' \models \beta$, since $\{\alpha\}S(\beta)$ is a valid triple. Then, $M' \models \beta'$, since $\beta \implies \beta'$. Finally, since the supports of $\alpha$ and $\alpha'$ as well as $\beta$ and $\beta'$ are equal, the validity of the Hoare triple holds.

Theorem 4.20 (Frame Rule Soundness). The Frame rule is sound.

Proof. First, we establish that for any $(M, H)$ such that $M \models \alpha \land \mu$ and $H = [Sp(\alpha \land \mu)]_M$, we never reach $\bot$. Consider $(M, H) \Rightarrow^* S$ as the sequence of configurations $P_2$. Construct the sequence of configurations $P_1$ as $(M, [Sp(\alpha)]_M) \Rightarrow^* S$, where each allocation from $S$ in $P_1$ chooses the same location to allocate as in $P_2$. We can show that for each step in $P_2$, there exists a corresponding step in $P_1$ such that:

1. at any corresponding step the allocated set on $P_2$ is a superset of the allocated set on $P_1$
2. the executions allocate and deallocate the same locations.

The claim as well as the first item is easy to show by structural induction on the program. Given that, the second is trivial, since a location available to allocate on $P_2$ is also available to allocate on $P_1$. Any location that is deallocated on $P_2$ that is unavailable on $P_1$ would cause $P_1$ to reach $\bot$, which is disallowed, since we are given that $\{\alpha\}S(\beta)$ is valid.

Thus, if we abort on the former, then we must abort on the latter, which is a contradiction, since we are given that $\{\alpha\}S(\beta)$ is valid. From the second item above, we can also establish that all mutations of the model are outside of $Sp(\mu)$, since it is unavailable on $P_1$ (we start with $Sp(\alpha)$ and allocate only outside $Sp(\alpha \land \mu) = Sp(\alpha) \cup Sp(\mu)$, and we are also given that the supports of $\alpha$ and $\mu$ are disjoint in any model). Therefore, if there exists a configuration $(M', H')$ such that $(M, H) \Rightarrow^* (M', H')$, then it must be the case that $M'$ is a mutation of $M$ that is stable on $Sp(\mu)$. Since $\{\alpha\}S(\beta)$ is valid, we have that $M' \models \beta$. Last, we conclude from the Frame Theorem (Theorem 3.7) that, since $M \models \mu$, $M' \models \mu$, which gives us $M' \models \beta \land \mu$.

We must also show that $H' = [Sp(\beta \land \mu)]_{M'}$. To show this, we can strengthen the inductive invariant above with the fact that at any corresponding step the allocated set on $P_2$ is not simply a superset of that on $P_1$, but in fact differs exactly by $Sp(\mu)$. This invariant establishes the desired claim, which concludes the proof of the frame rule.

4.8 Example

In this section, we verify an example program using the program logic developed above to demonstrate the utility of FL as a logic for annotating and reasoning with heap manipulating programs. We also offer some intuition about how our program logic can be deployed in a practical setting.

The following program performs an in-place reversal of the linked list pointed to by $i$:

```java
j := nil;
while (i != nil) do
  k := i.next;
  i.next := j;
  j := i;
  i := k;
```

A First-order Logic with Frames 7:31
Although we must show that \( j \) is the reverse of \( i \) for full functional correctness, we illustrate our program logic using a simpler contract, namely, that \( j \) points to a list and the end of the program. Full functional correctness can be proved using a similar proof structure, modeling the content of the linked lists as mathematical sequences. Linked lists modeled using the recursive definition list shown in Figure 4:

\[
\text{list}(x) := \text{ite}(x = \text{nil}, \top, \exists z : z = \text{next}(x). \text{list}(z) \land x \notin \text{Sp}(\text{list}(z))).
\]

We establish our desired contract using the following loop invariant that states that \( i \) and \( j \) point to disjoint lists: \( \text{list}(i) \land \text{list}(j) \land \text{Sp}(\text{list}(i)) \cap \text{Sp}(\text{list}(j)) = \emptyset \).

In the rest of this section, we prove that the above formula is indeed a loop invariant. Our proof uses a mix of both local and global rules from Sections 4.3 and 4.4. We also use the consequence rule along with the program rules in several places to simplify presentation. As a result, some detailed analysis is omitted, such as proving supports are disjoint to use the frame rule.

\[
\begin{align*}
&\{ \text{list}(i) \land \text{list}(j) \land \text{Sp}(\text{list}(i)) \cap \text{Sp}(\text{list}(j)) = \emptyset \land i \neq \text{nil} \} & \text{(consequence rule)} \\
&\{ \text{list}(i) \land \text{list}(j) \land \text{Sp}(\text{list}(i)) \cap \text{Sp}(\text{list}(j)) = \emptyset \land i \neq \text{nil} \land i \notin \text{Sp}(\text{list}(j)) \} & \text{(consequence rule: unfolding list definition)} \\
&\{ \exists k' : k' = \text{next}(i). \text{list}(k') \land i \notin \text{Sp}(\text{list}(k')) \land \text{list}(j) \land i \notin \text{Sp}(\text{list}(j)) \cap \text{Sp}(\text{list}(j)) = \emptyset \} & \text{(consequence rule: unfolding list definition)} \\
&\{ \exists k' : k' = \text{next}(i). \text{next}(i) = \text{next}(i) \land \text{list}(k') \land i \notin \text{Sp}(\text{list}(k')) \land \text{list}(j) \land i \notin \text{Sp}(\text{list}(j)) \cap \text{Sp}(\text{list}(j)) = \emptyset \} & \text{(consequence rule: unfolding list definition)} \\
&k := i. \text{next} \ ; & \text{(consequence rule, lookup-G rule)} \\
&\{ \text{next}(i) = \text{next}(i) \land \text{list}(k) \land i \notin \text{Sp}(\text{list}(k)) \land \text{list}(j) \land i \notin \text{Sp}(\text{list}(j)) \cap \text{Sp}(\text{list}(j)) = \emptyset \} & \text{(consequence rule, frame rule)} \\
&i. \text{next} := j \ ; & \text{(consequence rule, frame rule)} \\
&\{ \text{next}(i) = j \land \text{list}(k) \land i \notin \text{Sp}(\text{list}(k)) \land \text{list}(j) \land i \notin \text{Sp}(\text{list}(j)) \cap \text{Sp}(\text{list}(j)) = \emptyset \} & \text{(consequence rule: unfolding list definition)} \\
&\{ \text{list}(k) \land \text{next}(i) = j \land i \notin \text{Sp}(\text{list}(j)) \land \text{list}(j) \land \text{Sp}(\text{list}(k)) \cap \text{Sp}(\text{list}(j)) = \emptyset \} & \text{(consequence rule: unfolding list definition)} \\
&\{ \text{list}(k) \land \text{list}(i) \land \text{Sp}(\text{list}(k)) \cap \text{Sp}(\text{list}(i)) = \emptyset \} & \text{(mutation rule, frame rule)} \\
&j := i \ ; \ i := k \ ; & \text{(assignment-G rule)} \\
&\{ \text{list}(i) \land \text{list}(j) \land \text{Sp}(\text{list}(i)) \cap \text{Sp}(\text{list}(j)) = \emptyset \} & \text{(assignment-G rule)}
\end{align*}
\]

Using the loop invariant, proving \( j \) is a list after executing the full program above is then a simple application of the assignment, while, and consequence rules.

In the above proof, we applied the frame rule using the fact that \( i \) belongs neither to \( \text{Sp}(\text{list}(k)) \) nor \( \text{Sp}(\text{list}(j)) \). This disjointness condition can established using reasoning techniques for FO-RD formulas (see Section 6 for a discussion).

The reader may have also observed that the invariant we specified is precisely the intended meaning of \( \text{list}(i) \ast \text{list}(j) \) in separation logic. In fact, as we will see in Section 6, we can define a first-order macro \( \text{Star} \) as \( \text{Star}(\varphi, \psi) = \varphi \land \psi \land \text{Sp}(\varphi) \land \text{Sp}(\psi) = \emptyset \). We can then use this macro to succinctly represent disjoint supports while still retaining the flexibility afforded by Frame Logic.
See Section 6 for a broader discussion on the expressiveness and succinctness afforded by FL versus SL for stating similar properties.

5 EXPRESSING A PRECISE SEPARATION LOGIC

In this section, we show that FL is expressive by capturing a fragment of separation logic in frame logic; the fragment is a syntactic fragment of separation logic that defines only precise formulas—formulas that can be satisfied in at most one heaplet for any store. The translation also shows that frame logic can naturally and compactly capture such separation logic formulas.

5.1 A Precise Separation Logic

As discussed in Section 1, a crucial difference between separation logic and frame logic is that formulas in frame logic have uniquely determined supports/heaplets, while this is not true in separation logic (the heaplet for \( \alpha \lor \beta \) can be one that supports the truth of \( \alpha \) or one that supports the truth of \( \beta \) in Separation Logic). However, it is well known that in verification, determined heaplets are very natural (most uses of separation logic in fact are precise) and sometimes desirable. For instance, see Reference [15], where precision is used crucially in some proof rules regarding resource sharing to give sound semantics to concurrent separation logic, and Reference [53], where precise formulas are proposed in verifying modular programs as imprecision causes ambiguity in function contracts in the frame rule. The work in Reference [26] also relies on precise predicates in developing a Hoare logic for concurrent programs. Specifically, the conjunction rule is unsound in the context of imprecise predicates.

We define a fragment of separation logic that defines precise formulas (more accurately, we handle a slightly larger class inductively: formulas that, when satisfiable, have unique minimal heaplets for any given store).

Definition 5.1. PSL Fragment:

- \( sf \): formulas over the stack only (nothing dereferenced). Includes \( isatom() \), \( m(x) = y \) for immutable \( m \), \( \top \), background formulas, and Boolean combinations of these formulas.
- \( x \overset{f}{\rightarrow} y \)
- \( \text{ite}(sf, \varphi_1, \varphi_2) \) where \( sf \) is from the first bullet
- \( \varphi_1 \land \varphi_2 \) and \( \varphi_1 \ast \varphi_2 \)
- \( I \) where \( I \) contains all unary inductive definitions \( I \) that have unique heaplets inductively (list, tree, etc.). In particular, the body \( \rho_I \) of \( I \) is a formula in the PSL fragment \( (\rho_I[I \leftarrow \varphi] \) is in the PSL fragment provided \( \varphi \) is in the PSL fragment). Additionally, for all \( x \), if \( s, h \models I(x) \) and \( s, h' \models I(x) \), then \( h = h' \).
- \( \exists y. (x \overset{f}{\rightarrow} y) \ast \varphi_1 \).

Note that in the fragment negation and disjunction are disallowed, but mutually exclusive disjunction using \( \text{ite} \) is allowed. Existential quantification is only present when the topmost operator is \( \ast \) and where one of the formulas guards the quantified variable uniquely.

The semantics of this fragment follows the standard semantics of separation logic [21, 51, 52, 63], with the heaplet of \( x \overset{f}{\rightarrow} y \) taken to be \( \{x\} \). See Remark 1 in Section 3.2 for a discussion of a more accurate heaplet for \( x \overset{f}{\rightarrow} y \) being the set containing the pair \( (x, f) \) and how this can be modeled in the above semantics by using field-lookups using non-mutable pointers.

---

8While we only assume unary inductive definitions here, we can easily generalize this to inductive definitions with multiple parameters.
Theorem 5.2 (Minimum Heap). For any formula $\varphi$ in the PSL fragment, if there is an $s$ and $h$ such that $s, h \models \varphi$, then there is an $h_\varphi$ such that $s, h_\varphi \models \varphi$ and for all $h'$ such that $s, h' \models \varphi$, $h_\varphi \subseteq h'$.

5.2 Translation to Frame Logic

For a separation logic store and heap $s, h$ (respectively), we define the corresponding interpretation $M_{s, h}$ such that variables are interpreted according to $s$ and values of pointer functions on $\text{dom}(h)$ are interpreted according to $h$. For $\varphi$ in the PSL fragment, we first define a formula $P(\varphi)$, inductively, that captures whether $\varphi$ is precise. $\varphi$ is a precise formula iff, when it is satisfiable with a store $s$, there is exactly one $h$ such that $s, h \models \varphi$. The formula $P(\varphi)$ is in separation logic and will be used in the translation. To see why this formula is needed, consider the formula $\varphi_1 \land \text{ite}(sf, \varphi_2, \varphi_3)$. Assume that $\varphi_1$ is imprecise, $\varphi_2$ is precise, and $\varphi_3$ is imprecise. Under conditions where $sf$ is false, the heaplets for $\varphi_1$ and $\varphi_2$ must align. However, when $sf$ is false, the heaplets for $\varphi_1$ and $\varphi_3$ can be anything. Because we cannot initially know when $sf$ will be true or false, we need this separation logic formula $P(\varphi)$ that is true exactly when $\varphi$ is precise.

Definition 5.3. Precision predicate $P$:

- $P(sf) = \bot$ and $P(x \xrightarrow{f} y) = \top$
- $P(\text{ite}(sf, \varphi_1, \varphi_2)) = (sf \land P(\varphi_1)) \lor (\neg sf \land P(\varphi_2))$
- $P(\varphi_1 \land \varphi_2) = P(\varphi_1) \lor P(\varphi_2)$
- $P(\varphi_1 \land \varphi_2) = P(\varphi_1) \land P(\varphi_2)$
- $P(I) = \top$ where $I \in \mathcal{I}$ is an inductive predicate
- $P(\exists y. (x \xrightarrow{f} y) \land \varphi_1) = P(\varphi_1)$

Note that this definition captures precision within our fragment, since stack formulas are imprecise and pointer formulas are precise. The argument for the rest of the cases follows by simple structural induction. The interesting case is that of an inductive predicate $I$ whose body is assumed to be in the PSL fragment. We can show this by induction on the rank of $I(x)^9$ for an argument $x$ when $I(x)$ holds.

Now, we define the translation $T$ inductively:

Definition 5.4. Translation from PSL to Frame Logic:

- $T(sf) = sf$ and $T(x \xrightarrow{f} y) = (f(x) = y)$
- $\text{ite}(sf, \varphi_1, \varphi_2) = \text{ite}(T(sf), T(\varphi_1), T(\varphi_2))$
- $T(\varphi_1 \land \varphi_2) = T(\varphi_1) \land T(\varphi_2) \land T(P(\varphi_1)) \implies Sp(T(\varphi_2)) \subseteq Sp(T(\varphi_1))$
- $T(\varphi_1 \land \varphi_2) = T(\varphi_1) \land T(\varphi_2) \land Sp(T(\varphi_1)) \cap Sp(T(\varphi_2)) = \emptyset$
- $T(I) = T(\rho_I)$ where $\rho_I$ is the definition of the inductive predicate $I$ as in Section 3.
- $T(\exists y. (x \xrightarrow{f} y) \land \varphi_1) = \exists y : [f(x) = y] . [T(\varphi_1) \land x \notin Sp(T(\varphi_1))]$

Finally, recall that any formula $\varphi$ in the PSL fragment has a unique minimal heap (Theorem 5.2). With this (and a few auxiliary lemmas that can be found in Section 5.3), we have the following theorem, which captures the correctness of the translation:

Theorem 5.5. The following statements hold for any separation logic formula $\varphi$ in the PSL fragment:

---

9By the Knaster-Tarski theorem, we can compute the least-fixpoint by computing an increasing sequence pre-fixpoints for the kinds of definitions, we assume. This yields a “rank” for every $x$ such that $I(x)$ holds, namely, the index in the sequence of pre-fixpoints when $I(x)$ first holds.
(1) If \( s, h \models \phi \) for a store \( s \) and heap \( h \), then \( \mathcal{M}_{s,h} \models T(\phi) \).

(2) If \( \mathcal{M}_{s,h} \models T(\phi) \) for a frame model \( \mathcal{M}_{s,h} \), then \( s, h' \models \phi \), where \( h' \) is the interpretation of the support of \( T(\phi) \), i.e., \( \mathcal{M}_{s,h}(\text{Sp}(T(\phi))) \). Additionally, \( h' \) is minimal and is equal to \( h_\phi \) as in Theorem 5.2.

5.3 Frame Logic Can Capture the PSL fragment: Proofs

Lemma 5.6. For any formula \( \phi \) in the PSL fragment, if there is an \( s \) and \( h \) such that \( s, h \models \phi \) and we can extend \( h \) by some nonempty \( h' \) such that \( s, h \cup h' \models \phi \), then for any \( h'' \), \( s, h \cup h'' \models \phi \).

Proof. If a stack formula holds, then it holds on any heap. Pointer formulas and inductive definitions as defined can never have an extensible heap, so this is vacuously true.

For \( \text{ite}(sf, \varphi_1, \varphi_2) \), assume without loss of generality \( s, h \models sf \). Then, for any \( h' \), \( s, h' \models \varphi_1 \Leftrightarrow s, h' \models \text{ite}(sf, \varphi_1, \varphi_2) \). Then, use the induction hypothesis.

For \( \varphi_1 \land \varphi_2 \), for any \( h' \), \( s, h' \models \varphi_1 \land \varphi_2 \Leftrightarrow s, h' \models \varphi_1 \) and \( s, h' \models \varphi_2 \). If the conjoined formula can be extended, then both subformulas can be extended, and then we apply the induction hypothesis.

For separating conjunction, the nature of the proof is similar to conjunction, noting that the heap can be extended iff the heap of either subformula can be extended.

For existential formulas in our form, the proof is again similar, noting the heap is extensible iff the heap of \( \varphi_1 \) is extensible.

\[ \square \]

Theorem 5.2. For any formula \( \phi \) in the PSL fragment, if there is an \( s \) and \( h \) such that \( s, h \models \phi \), then there is an \( h_\phi \) such that \( s, h_\phi \models \phi \) and for all \( h' \) such that \( s, h' \models \phi \), \( h_\phi \subseteq h' \).

Proof. The minimal heaplets for stack formulas are empty. For \( x \xrightarrow{f} y \) the heaplet is uniquely \( \{x\} \).

For conjunction, there are three cases depending on if \( \varphi_1 \) or \( \varphi_2 \) or both have extensible heaplets.

We cover the most difficult case where they both have extensible heaplets here. By definition, we know \( s, h \models \varphi_1 \) and \( s, h \models \varphi_2 \). By induction, we know there are unique \( h_{\varphi_1} \) and \( h_{\varphi_2} \) such that \( h_{\varphi_1} \text{ and } h_{\varphi_2} \text{ model } \varphi_1 \text{ and } \varphi_2 \), respectively, and are minimal. Thus, \( h_{\varphi_1} \sqsubseteq h \text{ and } h_{\varphi_2} \sqsubseteq h \), so \( h_{\varphi_1} \cup h_{\varphi_2} \sqsubseteq h \). By Lemma 5.6, \( h_{\varphi_1} \cup h_{\varphi_2} \) is a valid heap for both \( \varphi_1 \) and \( \varphi_2 \). Thus, \( s, h_{\varphi_1} \cup h_{\varphi_2} \models \varphi_1 \land \varphi_2 \) and \( h_{\varphi_1} \cup h_{\varphi_2} \) is minimal.

For separating conjunction the minimal heaplet is (disjoint) union. For \( \text{ite} \), we pick the heaplet of either case, depending on the truth of the guard. By definition, inductive definitions will have minimal heaplets.

Inductive definitions have unique heaplets by the choice we made above and therefore vacuously satisfy the given statement.

For existentials, we know from the semantics of separation logic that every valid heap on a store \( s \) for the original existential formula is a valid heap for \( \psi' \equiv (x \xrightarrow{f} y) \ast \varphi_1 \) on a modified store \( s' \equiv s[y \mapsto v] \) for some \( v \). Since the constraint \( (x \xrightarrow{f} y) \) forces the value \( v \) to be unique, we can then invoke the induction hypothesis to conclude that the minimal heaplets of the existential formula on \( s \) and of \( \psi' \) on \( s' \) are the same. In particular, this means that existential formulas in our fragment also have a minimal heaplet.

\[ \square \]

Lemma 5.7. For any \( s, h \) such that \( s, h \models \phi \), we have \( \mathcal{M}_{s,h}(\text{Sp}(T(\phi))) = h_\phi \) where \( h_\phi \) is as above.

Proof. Structural induction on \( \phi \).

If \( \phi \) is a stack formula, then \( h_\phi = \text{Sp}(T(\phi)) = \emptyset \). If \( \phi \equiv x \xrightarrow{f} y \), then \( h_\phi = \text{Sp}(T(\phi)) = \{x\} \).
For \( \varphi \equiv \text{ite}(sf, \varphi_1, \varphi_2) \), because, \( s, h \models \varphi \), we know either \( s, h \models \varphi_1 \) or \( s, h \models \varphi_2 \), depending on the truth of \( sf \). Without loss of generality assume \( s, h \models sf \), then \( h_\varphi = h_{\varphi_1} \). Similarly, \( Sp(T(\varphi)) = Sp(sf) \cup Sp(T(\varphi_1)) = Sp(T(\varphi_1)) \) (heaplet of stack formulas is empty) and then we apply the induction hypothesis. Similarly if \( s, h \not\models sf \).

For \( \varphi \equiv \varphi_1 \land \varphi_2 \), we know from the proof of Theorem 5.2 that \( h_\varphi = h_{\varphi_1} \cup h_{\varphi_2} = M_{s,h}(Sp(T(\varphi_1))) \cup M_{s,h}(Sp(T(\varphi_2))) \). The guard parts of the translation \( Sp(\varphi) \), since they are all precise formulas that have empty heaplets.

For \( \varphi \equiv \varphi_1 \ast \varphi_2 \), the proof is the same to the previous case, again from the proof of Theorem 5.2.

For an inductive definition \( I \), recall that \( \rho_I[I \leftarrow \varphi] \) is in the PSL fragment (and crucially does not mention \( I \)). Assume \( \varphi \) is fresh and does not occur in \( \rho_I \). Define \( \rho_I' \equiv \rho_I[I \leftarrow \varphi] \) and note that \( \rho_I = \rho_I'[\varphi \leftarrow I] \). This means that \( h_{\rho_I} = h_{\rho'_I}[h_\varphi \leftarrow h_I] \). We also see that \( Sp(T(\rho_I)) = Sp(T(\rho'_I))[Sp(T(\varphi)) \leftarrow Sp(T(\rho_I))] \). Because \( h_{\rho'_I} = Sp(T(\rho'_I)) \) (by the other cases in this proof and, since \( \rho'_I \) does not mention \( I \)), we see the heaplets are related by the same sets of recursive equations and we are done.

For existentials, we have from the definition of the \( Sp \) operator that the support of the translation of the existential formula is the same as that of \( \{x\} \cup Sp(T(\varphi_1)) \). The claim then follows from the definition of heaplet of existentials in separation logic as well as the inductive hypothesis for \( \varphi_1 \).

\[ \square \]

**Theorem 5.5.** The following statements hold for any separation logic formula \( \varphi \) in the PSL fragment:

1. If \( s, h \models \varphi \) for a store \( s \) and heap \( h \), then \( M_{s,h} \models T(\varphi) \).
2. If \( M_{s,h} \models T(\varphi) \) for a frame model \( M_{s,h} \), then \( s, h' \models \varphi \), where \( h' \) is the support of \( T(\varphi) \), i.e., \( M_{s,h}(Sp(T(\varphi))) \).

**Proof.** First statement: Structural induction on \( \varphi \).

If \( \varphi \) is a stack formula or a pointer formula, then this is true by construction. If \( \varphi \) is an if-then-else formula, then the claim is true by construction and the induction hypothesis.

If \( \varphi = \varphi_1 \land \varphi_2 \), then we know by the induction hypothesis that \( M_{s,h} \models T(\varphi_1) \) and \( M_{s,h} \models T(\varphi_2) \). Further, from the semantics of separation logic, we have that if \( \varphi_1 \) is precise, then \( h_{\varphi_1} = h \). Therefore, \( h_{\varphi_2} \subseteq h_{\varphi_1} \) (by Lemma 5.2). Therefore, from Lemma 5.7, we have that \( M_{s,h} \models Sp(T(\varphi_2)) \subseteq Sp(T(\varphi_1)) \). Similarly if \( \varphi_2 \) is precise. This justifies the two latter conjuncts of the translation.

If \( \varphi = \varphi_1 \ast \varphi_2 \), then we know there exist \( h_1, h_2 \) such that \( h_1 \cap h_2 = \emptyset \) and \( s, h_1 \models \varphi_1 \) and \( s, h_2 \models \varphi_2 \). Then, from Lemma 5.2, we have that \( h_{\varphi_1} \subseteq h_1 \) and \( h_{\varphi_2} \subseteq h_2 \). Thus, by Lemma 5.7, we have that \( M_{s,h} \models Sp(T(\varphi_1)) \cap Sp(T(\varphi_2)) = \emptyset \). The other conjuncts follow from the induction hypothesis.

Similarly to the proof of Lemma 5.7, we can show that the translation of the inductive definition satisfies the same recursive equations as the original inductive definition and we are done.

If \( \varphi \) is an existential, then the result follows from definition and the induction hypothesis.

Second statement: Structural induction on \( \varphi \).

By construction, induction hypotheses, and Lemma 5.7, all cases can be discharged besides conjunction and inductive predicates.

For conjunction, if \( \varphi = \varphi_1 \land \varphi_2 \), then we have from the induction hypothesis that \( s, h_{\varphi_1} \models \varphi_1 \) and \( s, h_{\varphi_2} \models \varphi_2 \). If \( \varphi_1 \) is precise, then we know \( M_{s,h} \models Sp(T(\varphi_2)) \subseteq Sp(T(\varphi_1)) \) and therefore \( h_{\varphi_2} \subseteq h_{\varphi_1} \) (from Lemma 5.7). Similarly, if \( \varphi_2 \) is precise, then \( M_{s,h} \models Sp(T(\varphi_1)) \subseteq Sp(T(\varphi_2)) \) as well as \( h_{\varphi_1} \subseteq h_{\varphi_2} \). In particular, if they are both precise, then their supports (and therefore minimal heaplets) are equal, and \( h' = h_{\varphi_1} \cup h_{\varphi_2} \) (from the proof of Lemma 5.2) = \( h_{\varphi_1} = h_{\varphi_2} \), and we are done. If only \( \varphi_1 \) is precise (similarly if only \( \varphi_2 \) is precise), then we have as above that \( h_{\varphi_2} \subseteq h_{\varphi_1} \) and \( h_{\varphi_1} = h' \). Moreover, we know by Lemma 5.6 that \( s, h_{\varphi_1} \models \varphi_2 \) and we are done. If neither is
precise, then both heaps are extensible, so we know by Lemma 5.6 that \( s, h_{\varphi_1} \cup h_{\varphi_2} \models \varphi_1 \) and \( s, h_{\varphi_1} \cup h_{\varphi_2} \models \varphi_2 \) and we are done.

For \( \varphi \) an inductive predicate, we know that \( M_{s,h}[Sp(T(\varphi))] \models T(\varphi) \). The remainder follows, since, because we restrict the form of inductive predicates to have a unique heap at each level, the translated inductive predicate will satisfy the same recursive equations as \( \varphi \).

\[ \square \]

### 6 DISCUSSION

#### 6.1 Comparison with Separation Logic

Design. The design of frame logic is, in many ways, inspired by the design choices of separation logic. Separation logic formulas implicitly hold on tight heaplets—models are defined on pairs \((s, h)\), where \( s \) is a store (an interpretation of variables) and \( h \) is a heaplet that defines a subset of the heap as the domain for functions/pointers. In Frame Logic, we choose to not define satisfiability with respect to heaplets but define it with respect to the entire global heap. However, we give access to the implicitly defined heaplet using the operator \( Sp \) and give a logic over sets to talk about supports. The separating conjunction operation \(*\) can then be expressed using normal conjunction of the two formulas and a constraint that says that the supports of the formulas are disjoint, e.g., \( x \xrightarrow{\text{next}} y * \text{list}(y) \) can be expressed in FL as "next\((x) = y \land \text{list}(y) \land Sp(\text{next}(x) = y) \land Sp(\text{list}(y)) = \emptyset\)."

Crucially, we do not allow formulas to have multiple supports \((Sp \) is a function\). Precise fragments of separation logic have been proposed and accepted in the separation logic literature a way of giving robust semantics in handling modular functions and concurrency [15, 53]. Section 5 details a translation of a precise fragment of separation logic (with \(*\) but not magic wand) to frame logic that shows the natural connection between precise formulas in separation logic and frame logic. When converting arbitrary (potentially imprecise) separation logic formulas to first-order logic, there is an inherent existential quantifier over heaplets that makes reasoning difficult due to the non-uniqueness of heaplets. Our logic, however, does not have this problem, because supports are uniquely defined, and hence we do not need to quantify over heaplets (as in the above example).

Frame logic, through the support operator, facilitates local reasoning much in the same way as separation logic does, and the frame rule in frame logic supports frame reasoning in a similar way as the frame rule in separation logic. The key difference between frame logic and separation logic is the adherence to a first-order logic (with recursive definitions) both in terms of syntax and expressiveness. Note that there is no difference in regards to completeness, as both Separation Logic and FO-RD are incomplete [17, 45, 63]. We refer the reader to page 13 where several nuances in the design of FL semantics are explained.

Reasoning and Automation. There are several other key differences between separation logic and frame logic. First, in separation logic, the magic wand operator is needed to express the weakest precondition [63]. Consider, for example, computing the weakest precondition of the formula \( \text{list}(x) \) with respect to the code \( y.n \xleftarrow{n} z \). The weakest precondition should essentially describe the (tight) heaplets such that changing the \( n \) pointer from \( y \) to \( z \) results in \( x \) pointing to a list. In separation logic, this is expressed typically (see Reference [63]) using magic wand as \( (y \xrightarrow{n} z) \iff \text{list}(x) \). However, the magic wand operator is inherently a second-order operator [14]. The formula \( \alpha \rightarrow \beta \) holds on a heaplet \( h \) if for any disjoint heaplet that satisfies \( \alpha, \beta \) will hold on the conjoined heaplet. Expressing this property (for arbitrary \( \alpha \), whose heaplet can be unbounded) requires quantifying over unbounded heaplets satisfying \( \alpha \), which is not first-order expressible.

In frame logic, we instead rewrite the recursive definition \( \text{list}(\cdot) \) to a new one \( \text{list}'(\cdot) \) that captures whether \( x \) points to a list, assuming that \( n(y) = z \) (see Section 4.4). This property continues
to be expressible in frame logic and can be converted to first-order logic with recursive definitions (see Section 3.6). Note that we are exploiting the fact that there is only a bounded amount of change to the heap in loop-free programs to express this in FL.

This is significant, as FO-RD is weaker than second-order logic.\textsuperscript{10} In terms of automation, little is known about automating second-order logic, whereas automation for FOL and FO-RD is ubiquitous in prior literature. We discuss some possible approaches to automating FL in Section 6.2. In contrast, even automation specific to SL seems hard, and indeed many state-of-the-art solvers that automate reasoning with SL do not support the magic wand [64].

**Expressiveness and Succinctness.** We now turn to expressiveness and succinctness. In separation logic, separation of structures is expressed using $\ast$, and in frame logic, such a separation is expressed using conjunction and an additional constraint that says that the supports of the two formulas are disjoint. A precise separation logic formula of the form $\alpha_1 \ast \alpha_2 \ast \ldots \alpha_n$ is succinct and would get translated to a much larger formula in frame logic, as it would have to state that the supports of each pair of formulas are disjoint. We believe this can be tamed using macros $(\text{Star}(\alpha, \beta) = \alpha \land \beta \land \text{Sp}(\alpha) \cap \text{Sp}(\beta) = \emptyset)$.

There are, however, several situations where frame logic leads to more compact and natural formulations. For instance, consider expressing the property that $x$ and $y$ point to lists, which may or may not overlap. In Frame Logic, we simply write $\text{list}(x) \land \text{list}(y)$. The support of this formula is the union of the supports of the two lists. In separation logic, we cannot use $\ast$ to write this compactly (while capturing the tightest heaplet). Note that the formula $(\text{list}(x) \ast \text{true}) \land (\text{list}(y) \ast \text{true})$ is not equivalent, as it would also hold in heaplets that are larger than the set of locations of the two lists. The simplest formulation we know is to write a recursive definition $\text{lseg}(u, v)$ for list segments from $u$ to $v$ and use quantification: $(\exists z. \text{lseg}(x, z) \ast \text{lseg}(y, z) \ast \text{list}(z)) \lor (\text{list}(x) \ast \text{list}(y))$ where the definition of $\text{lseg}$ is the following: $\text{lseg}(u, v) \equiv (u = v \land \text{emp}) \lor (\exists w. u \to w \ast \text{lseg}(w, v))$.

If we wanted to say $x_1, \ldots, x_n$ all point to lists, that may or may not overlap, then in FL we can say $\text{list}(x_1) \land \text{list}(x_2) \land \cdots \land \text{list}(x_n)$. However, in separation logic, the simplest way seems to be to write using $\text{lseg}$ and a linear number of quantified variables and an exponentially-sized formula. Now, consider the property saying $x_1, \ldots, x_n$ all point to binary trees, with pointers left and right, and that can overlap arbitrarily. We can write it in FL as $\text{tree}(x_1) \land \cdots \land \text{tree}(x_n)$, while a formula in separation logic that expresses this property seems very complex.

The difficulty of using separation logic to capture such overlapping datastructures [39, 41] has been observed in prior literature. For example, the work in Reference [28] introduces an overlapping conjunction operator $\cup$ to separation logic, which, unlike $\ast$ or $\land$, allows the conjunction of two properties whose heaplets may overlap but not coincide and can express the above properties we mention succinctly. However, the ramification rules proposed in the same work to reason with overlapping datastructures also seem inherently second-order, introducing the magic wand [14]. The work in Reference [54] explores a decidable fragment of separation logic with inductive definitions including guarded magic wands to capture overlaid datastructures.

In summary, frame logic is a logic that is built on similar principles as SL that supports frame reasoning in similar ways but offers in addition the benefit of being translatable to first-order logic. This enables the usage of existing automation for FO-RD for reasoning about programs.

\textsuperscript{10}We know from classic results in finite model theory that FO+$\text{lfp}$ captures the complexity class P (polynomial time) [29, 70], while second order logic (SO) captures the entire polynomial hierarchy (well above NP). Hence, FO+$\text{lfp}$ is strictly weaker than second-order logic (unless the polynomial hierarchy collapses to P). One could argue that FO-RD with sets is closer to monadic second order logic (MSO), where quantification over sets is universal; but even then MSO is less powerful than SO. For instance, on linear structures with finite labels, MSO captures regular languages only, while SO is much more powerful and can encode Turing machine executions.
annotated with FL formulas, as FL is also closed under weakest preconditions (avoiding second-order operators the magic wand). Finally, the choices made in defining the syntax and semantics of FL allow expressing certain properties more naturally and compactly than SL, while others may only be expressible more verbosely without macros or other additional syntax.

6.2 Reasoning with Frame Logic

An important advantage in using frame logic is that it is translatable to a first-order logic with recursive definitions, opening up the possibility of using solvers for FO-RD to reason with frame logic.

Notably, we are inspired by the reasoning using the framework on Natural Proofs [45, 57, 62]. In this work, recursive definitions with least fixpoint semantics are first abstracted to fixpoint definitions, lending to a formulation in pure first-order logic. Second, universal quantifiers are instantiated using terms in the formula, and recursive definitions are unfolded on terms as well. Finally, the resulting formulas, which are quantifier-free, are checked using SMT solvers. A remarkable result is that for a safe fragment of the logic (which verification conditions for datastructure manipulating programs often adhere to), the above technique is complete with respect to pure FO reasoning [45]. The work in Reference [48] also attempts to bridge the gap between the first-order reasoning power of natural proofs and the least-fixpoint semantics of inductive definitions by synthesizing inductive lemmas.

Our main hope of reasoning with frame logic resides in this technique, which has already been used in reasoning with Dryad, a precise fragment of separation logic, by converting it to first-order logic with recursive definitions [57, 62]. The evaluation of Dryad on a suite of 150 heap programs has been reported in the works on Natural Proofs, providing evidence that the technique works well in practice. However, it is possible that verification conditions derived using the weakest precondition rules developed in this article may introduce quantification that makes reasoning harder than necessary. We believe that an approach based on verification condition generation using strongest postconditions (or symbolic execution), as in the works in References [57, 62], combined with restrictions on frame logic annotations that prevent the introduction of existential quantification in specifications would lead to an efficient solver. In particular, the support of recursively defined functions can be generated automatically using our translation, while the work in References [57, 62] translated heaplets for Dryad formulas using manual, unverified translations. However, this work is beyond the scope of this article, and we leave it for the future. The main technical challenge is to formulate restrictions on frame logic annotations so they do not use existential quantification and realize supports of inductively defined datastructures as their own inductive definitions (that compute sets of locations), again without introducing additional quantifiers.

Another mechanism to reason with frame logic is to convert it into FO-RD and use it in frameworks such as Dafny [42] or Boogie [3, 7], which support rich user-annotation based reasoning. In particular, the methodology of region logic [4–6] suggests ways of encoding supports as sets in first-order logic, which are then manipulated using ghost annotations that update a ghost state and help prove theorems (contracts). This methodology typically requires help from the user (i.e., the ghost annotations), but much of the reasoning is delegated to automatic logic engines (mainly SMT solvers). Region logic [4–6] itself is a logic for heap manipulating programs that supports explicit heaplets and frame reasoning over them, rather than the implicit heaplets of frame logic.

\[ \text{See http://madhu.cs.illinois.edu/vcdryad/}. \]
7 RELATED WORK

The frame problem \cite{12, 27} is an important problem in many different domains of research. In the broadest form, it concerns representing and reasoning about the effects of a local action without requiring explicit reasoning regarding static changes to the global scope. In the domain of artificial intelligence, for example, one wants a logic that can seamlessly state and reason that if a door is opened in a lit room, then the lights continue to stay turned on. This issue is present in the domain of verification as well with heap-manipulating programs in reasoning about heap updates.

Many solutions have been proposed to this problem. The most prominent proposal in the verification context is separation logic \cite{21, 51, 52, 63}, which we discussed in detail in the previous section.

The work on Dynamic Frames \cite{35, 36} and similarly inspired approaches such as Region Logic \cite{4–6} allow programs to explicitly specify parts of the heap that may be modified. The key idea is the notion of *regions*, which are subsets of locations that can be manipulated in code (as ghost variables) as well as used in annotations to perform frame reasoning. This allows a finer-grained specification of the portion of the heap modified by programs, avoiding special symbols such as $\ast$ and $\neg\ast$. Section 6.2 discusses verification methodologies based on these logics as well as the idea of using these techniques for reasoning with FL. In contrast, FL itself has *implicit* supports given using the $Sp$ operator.

The work on Implicit Dynamic Frames \cite{43, 56, 65, 66} bridges the worlds of separation logic and dynamic frames—it uses separation logic and fractional permissions to implicitly define frames (reducing annotation burden), allows annotations to access these frames, and translates them into regions (sets) for first-order reasoning. Our work is similar in that frame logic also implicitly defines regions and gives annotations access to these regions and can be easily translated to pure FO-RD for first-order reasoning. However, implicit dynamic frames do not allow a formula to access anything in a function that is not in the support of the function’s precondition. Further, all implementations of Implicit Dynamic Frames, such as Chalice \cite{56} and Viper \cite{47}, only handle restricted formulas. For example, non-separating conjunction is prohibited. Translations from the VeriFast theorem prover to Implicit Dynamic Frames \cite{33, 34} contain similarly restricted formulas.

One of the key distinctions with separation logic is the property of unique heaplets for FL formulas. Determined heaplets have been used \cite{53, 57, 62}, as they are more amenable to automated reasoning. In particular, a separation logic fragment with determined heaplets known as precise predicates is defined in Reference \cite{53}, which we capture using frame logic in Section 5.

There is also a rich literature on reasoning with these heap logics for program verification. Decidability is an important dimension, and there is a lot of work on decidable logics for heaps with separation logic specifications \cite{8–10, 20, 50, 58}. The work based on EPR (Effectively Propositional Reasoning) for specifying heap properties \cite{30–32} provides decidability, as does some of the work that translates separation logic specifications into classical logic \cite{59}.

The work in Reference \cite{11} defines an extension of first-order logic and inductive definitions with footprints, where the design decisions on the semantics of footprints are very similar to ours. The work develops the notion of “separation predicates” based on these footprint expressions as well as verification conditions for programs annotated with first-order formulas involving separation predicates. The work also realizes an implementation of this technique by generating verification conditions and using Why3 \cite{24, 25}. The footprint expressions compute the same set as our $Sp$ operator (with some minor syntactic changes), but our work handles a much larger fragment. In particular, apart from only defining supports for recursive predicates, the work in Reference \cite{11} disallows quantification, non-separating conjunction, and separation predicates in the body.
of recursive predicates. This makes it difficult to define many of the typical data-structures shown in Figure 4 in their logic.

Translating separation logic into other logics and reasoning with them is another solution pursued in a lot of recent efforts [19, 45, 46, 57, 57, 59–62, 67]. Other techniques including recent work on cyclic proofs [16, 68] use heuristics for reasoning about recursive definitions.

Finally, there is also work that studies first-order logics such as FO(ID) with more general non-monotonic inductive definitions motivated by AI applications such as the frame problem and the modeling of causal processes of knowledge [22, 23]. FO(ID) generalizes the least-fixpoint semantics [69] we use for definitions in FO-RD (which are all required to be monotonic).

8 CONCLUSIONS

Our main contribution is the development of Frame Logic, a first-order logic endowed with an explicit operator $S_p$ that recovers the implicit supports of formulas and supports frame reasoning. We have argued its expressiveness by capturing several properties of data-structures naturally and succinctly and by showing that it can express a precise fragment of separation logic. The program logic built using frame logic supports local heap reasoning, frame reasoning, and weakest tightest preconditions across loop-free programs.

We believe that frame logic is an attractive alternative to separation logic, built using similar principles as separation logic while staying within the first-order logic world. The first-order nature of the logic makes it potentially amenable to easier automated reasoning.

Practical realization of a tool for verifying programs in a standard programming language with frame logic annotations by marrying it with existing automated techniques and tools for first-order logic (in particular, References [37, 46, 57, 62, 67]) is the most compelling future work.

Other avenues for future work involve extensions to Frame Logic. For example, the work in Reference [40] uses magic wand for verifying certain design patterns including iterators, and extending Frame Logic similarly is an interesting possible direction. Other potential extensions include support for function pointers and permission models.

REFERENCES

[1] 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 (POPL’79). Association for Computing Machinery, New York, NY, 110–119. DOI: https://doi.org/10.1145/567752.567763

[2] Krzysztof R. Apt. 1981. Ten years of Hoare’s logic: A survey—part I. ACM Trans. Program. Lang. Syst. 3, 4 (Oct. 1981), 431–483. DOI: https://doi.org/10.1145/357146.357150

[3] Anindya Banerjee, Mike Barnett, and David A. Naumann. 2008. Boogie meets regions: A verification experience report. In Verified Software: Theories, Tools, Experiments, Natarajan Shankar and Jim Woodcock (Eds.). Springer Berlin, 177–191.

[4] Anindya Banerjee and David A. Naumann. 2013. Local reasoning for global invariants, part II: Dynamic boundaries. J. ACM 60, 3 (June 2013). DOI: https://doi.org/10.1145/2485981

[5] 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, 387–411.

[6] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2013. Local reasoning for global invariants, part I: Region Logic. J. ACM 60, 3 (June 2013). DOI: http://doi.acm.org/10.1145/2485982

[7] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In Formal Methods for Components and Objects, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin, 364–387.

[8] 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 (FMCO’05). Springer-Verlag, Berlin, 115–137. DOI: https://doi.org/10.1007/11804192_6

[9] 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 (FSTTCS’04). Springer-Verlag, Berlin, 97–109. DOI: https://doi.org/10.1007/978-3-540-30538-5_9

ACM Transactions on Programming Languages and Systems, Vol. 45, No. 2, Article 7. Publication date: May 2023.


[10] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic execution with separation logic. In Proceedings of the 3rd Asian Conference on Programming Languages and Systems (APLAS’05). Springer-Verlag, Berlin, 52–68. DOI: https://doi.org/10.1007/11575467_5

[11] François Bobot and Jean-Christophe Filliâtre. 2012. Separation predicates: A taste of separation logic in first-order logic. In Formal Methods and Engineering, Toshiaki Aoki and Kenji Taguchi (Eds.). Springer Berlin, 167–181.

[12] Alex Borgida, John Mylopoulos, and Raymond Reiter. 1995. On the frame problem in procedure specifications. IEEE Trans. Softw. Eng. 21, 10 (Oct. 1995), 785–798. DOI: https://doi.org/10.1109/32.469460

[13] K. Brinck and N. Y. Foo. 1981. Analysis of algorithms on threaded trees. Comput. J. 24, 2 (01 1981), 148–155. DOI: https://doi.org/10.1093/comjnl/24.2.148

[14] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. 2008. On the almighty wand. In Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS’01). Springer-Verlag, Berlin, 108–119.

[15] Stephen Brookes. 2007. A semantics for concurrent separation logic. Theor. Comput. Sci. 375, 1-3 (Apr. 2007), 227–270. DOI: https://doi.org/10.1016/j.tcs.2006.12.034

[16] 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, 131–146. Retrieved from http://dl.acm.org/citation.cfm?id=2032266.2032278

[17] Cristiano Calcagno, Hongseok Yang, and Peter W. O’Hearn. 2001. Computability and complexity results for a spatial assertion language for data structures. In Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS’01). Springer-Verlag, Berlin, 108–119.

[18] 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 (FCS’80). IEEE Computer Society, 333–347. DOI: https://doi.org/10.1109/SFCS.1980.41

[19] 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, 307–320. DOI: https://doi.org/10.1109/ICECCS.2007.17

[20] 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 (CONCUR’11). Springer-Verlag, Berlin, 235–249.

[21] Stéphane Demri and Morgan Deters. 2015. Separation logics and modalities: A survey. In Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR’14). AAAI Press, 22–31.

[22] Marc Denecker. 2000. Extending classical logic with inductive definitions. In Proceedings of the 1st International Conference on Computational Logic (CL’00). Springer-Verlag, Berlin, 703–717.

[23] Marc Denecker and Joost Vennekens. 2014. The well-founded semantics is the principle of inductive definition, revisited. In Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR’14). AAAI Press, 22–31.

[24] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where programs meet provers. In Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP’13). Springer-Verlag, Berlin, 125–128. DOI: https://doi.org/10.1007/978-3-642-37036-6_6

[25] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where programs meet provers. In Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP’13). Springer-Verlag, Berlin, 125–128. DOI: https://doi.org/10.1007/978-3-642-37036-6_8

[26] Alexey Gotsman, Josh Berdine, and Byron Cook. 2011. Precision and the conjunction rule in concurrent separation logic. In Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP’13). Springer-Verlag, Berlin, 125–128. DOI: https://doi.org/10.1007/978-3-642-37036-6_8

[27] Patrick J. Hayes. 1981. The frame problem and related problems in artificial intelligence. In Readings in Artificial Intelligence, Bonnie Lynn Webber and Nils J. Nilsson (Eds.). Morgan Kaufmann, 223–230. DOI: https://doi.org/10.1016/B978-0-934613-03-3.50020-9

[28] 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 (POPL’13). Association for Computing Machinery, New York, NY, 523–536. DOI: https://doi.org/10.1145/2429069.2429131

[29] Neil Immerman. 1982. Relational queries computable in polynomial time (extended abstract). In Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS’82). Association for Computing Machinery, New York, NY, 147–152. DOI: https://doi.org/10.1109/SFCS.1982.40

[30] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanoski, and Mooly Sagiv. 2014. Modular reasoning about heap paths via effectively propositional formulas. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’14). ACM, New York, NY, 385–396. DOI: https://doi.org/10.1145/2535838.2535854

ACM Transactions on Programming Languages and Systems, Vol. 45, No. 2, Article 7. Publication date: May 2023.
[31] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanescu, and Mooly Sagiv. 2013. Effectively-propositional reasoning about reachability in linked data structures. In Proceedings of the 25th International Conference on Computer Aided Verification (CAV’13). Springer-Verlag, Berlin, 756–772. DOI: https://doi.org/10.1007/978-3-642-39799-8_53

[32] Shachar Itzhaky, Nikolaj Bjørner, Thomas Reps, Mooly Sagiv, and Aditya Thakur. 2014. Property-directed shape analysis. In Proceedings of the 16th International Conference on Computer Aided Verification (CAV’14). Springer-Verlag, Berlin, 35–51. DOI: https://doi.org/10.1007/978-3-319-08867-9_3

[33] 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 3rd International Conference on NASA Formal Methods (NFM’11). Springer-Verlag, Berlin, 41–55.

[34] 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, 202–221.

[35] 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, 268–283.

[36] I. T. Kassios. 2011. The dynamic frames theory. Form. Asp. Comput. 23, 3 (May 2011), 267–288. DOI: https://doi.org/10.1007/s00165-010-0152-5

[37] Laura Kovács, Simon Robillard, and Andrei Voronkov. 2017. Coming to terms with quantified reasoning. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17). ACM, New York, NY, 260–270. DOI: https://doi.org/10.1145/3009837.3009887

[38] Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and Vampire. In Computer Aided Verification, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin, 1–35.

[39] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020. Local reasoning for global graph properties. In Programming Languages and Systems, Peter Müller (Ed.). Springer International Publishing, Cham, 308–335.

[40] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. 2009. Program analysis for overlaid data structures. In Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583 (VMCAI 2016). Springer, Berlin, 202–221.

[41] Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin, 202–221.

[42] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. 2011. Program analysis for overlaid data structures. In Computer Aided Verification, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin, 592–608.

[43] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR’10). Springer-Verlag, Berlin, 348–370. DOI: https://doi.org/10.5555/1939141.1939161

[44] 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, 378–393. DOI: https://doi.org/10.1007/978-3-642-00590-9_27

[45] Leonid Libkin. 2004. Elements of Finite Model Theory. Springer. DOI: https://doi.org/10.1007/978-3-642-06008-3

[46] Christof Löding, P. Madhusudan, and Lucas Peña. 2018. Foundations for natural proofs and quantifier instantiation. Proc. ACM Program. Lang. 2, POPL (2018), 10:1–10:30. DOI: https://doi.org/10.1145/3158098

[47] P. Madhusudan, Xiaokang Qiu, and Andrei Ştefănescu. 2012. Recursive proofs for inductive tree data-structures. In Proceedings of the 39th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’12). ACM, New York, NY, 105–116. DOI: https://doi.org/10.1145/1401861.1481874

[48] 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. DOI: https://doi.org/10.3233/978-1-61499-028-4-286
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, 1–19. Retrieved from 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 (POPL’04). ACM, New York, NY, 268–280. DOI: https://doi.org/10.1145/964001.964024.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.

Jens Pagel. 2020. Decision Procedures for Separation Logic: Beyond Symbolic Heaps. Ph.D. Dissertation. Wien.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05). ACM, New York, NY, 247–258. DOI: https://doi.org/10.1145/1040305.1040326.

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, 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 (PLDI’14). ACM, New York, NY, 440–451. DOI: https://doi.org/10.1145/2594291.2594325.

Jianzhi Yuan, Antonio Pérez and Andrey Rybalchenko. 2013. Separation logic modulo theories. In Programming Languages and Systems, Gilles Barthe (Ed.). Springer Berlin, 439–458.