Matching Plans for Frame Inference in Compositional Reasoning

Andreas Lööw
Imperial College London, UK

Daniele Nantes-Sobrinho
Imperial College London, UK

Sacha-Élie Ayoun
Imperial College London, UK

Petar Maksimović
Imperial College London, UK
Runtime Verification Inc., Chicago, IL, USA

Philippa Gardner
Imperial College London, UK

Abstract

The use of function specifications to reason about function calls and the manipulation of user-defined predicates are two essential ingredients of modern compositional verification tools based on separation logic. To execute these operations successfully, these tools must be able to solve the frame inference problem, that is, to understand which parts of the state are relevant for the operation at hand. We introduce *matching plans*, a concept that is used in the Gillian verification platform to automate frame inference efficiently. We extract matching plans and their automation machinery from the Gillian implementation and present them in a tool-agnostic way, making the Gillian approach available to the broader verification community as a verification-tool design pattern.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Separation logic; Theory of computation → Automated reasoning

Keywords and phrases Compositional reasoning, separation logic, frame inference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.26

Funding This work was supported by the EPSRC Fellowship "VetSpec: Verified Trustworthy Software Specification" (EP/R034567/1).

Acknowledgements We would like to thank José Fragoso Santos, who conceived the idea of matching plans (which we formalise in this work) as part of the work on JaVerT [6] and did their original implementation in Gillian. We would also like to thank the anonymous reviewers for their comments.

1 Introduction

Separation logic [18, 21] has enabled the verification community to develop analyses and tools that are *compositional* in the sense that they are able to analyse parts of the program in isolation and reuse the obtained results in broader contexts. Currently, some of the most prominent such tools are VeriFast [7], Viper [16], Gillian [13], and CN [20]. These tools achieve compositionality by being able to use function specifications at call sites instead of executing function bodies. In addition, to be able to reason about data structures such as lists and trees, the tools include support for user-defined inductive predicates that describe these data structures. When using function specifications and manipulating predicates, the tools have to be able to solve the *frame inference problem* [3], that is, understand which part of the state is relevant for the operation that is being performed. The ability to handle this problem efficiently is essential for their scalability and usability.

As part of building tools for compositional analysis, it is up to the tool designers to choose how to tackle frame inference, and the implications of the associated decisions should be understood closely as they affect both the tool implementation and the user experience. For example, two important tool-design questions are: "Which specification language should the tool use?" and "Is there a particular style in which the specifications should be written?". Expectedly, the approaches in the literature are many and varied (which we discuss further in §7).

In this paper, we present the approach of the Gillian verification tool to the frame inference problem and show how the choices made allow for *efficient and predictable automation*. The approach captures and automates the *assertion-adaptation workflow* that users must follow to facilitate frame inference when working with tools that offer less automation, such as VeriFast and Viper. In more detail, for the assertions of function specifications and predicate definitions, Gillian automatically constructs a *matching plan* (MP), which provides:

1. (*efficiency*) an ordering of the subcomponents of each assertion (technically: the simple assertions) that guarantees that the associated frame inference will not backtrack, together with a description of how all associated free and existentially quantified variables can be learnt; and
2. (*predictability*) a clean separation between the structural and computational portions of frame inference.

MPs and their construction have not been described in depth before; we formalise both in a tool-agnostic way, thereby making the Gillian approach available to the broader verification community as a verification-tool design pattern.

The paper is structured as follows. We first introduce MPs informally using examples and discuss the key insights in §2. We then establish the required preliminaries in §3 and introduce MPs formally in §4, focusing on a core illustrative fragment of MPs as implemented in Gillian. Next, in §5, we show how to extend the MPs of §4 with more complex features that are available in Gillian. Finally, we conclude by evaluating the scalability and performance of the MP-based automation of Gillian (§6) and giving a detailed comparison with related work (§7).

2 Overview

We give an informal overview of *matching plans* (MPs), the key new concept we introduce in this paper. We first introduce the required background, which is the *consume/produce engine architecture* utilised by modern compositional symbolic execution tools, including VeriFast, Viper, and Gillian. Then, with the background in place, we introduce MPs using examples.

2.1 Background: Symbolic Execution Based on Consume and Produce

We place ourselves in the setting of semi-automated compositional verification tools based on symbolic execution [1] and separation logic (SL) and are underpinned by SMT solvers. In this context, the frame inference problem amounts to, given an assertion and a symbolic state, understanding which part of the symbolic state corresponds to the assertion. In particular, we are interested in tools such as VeriFast, Viper, and Gillian, implemented using *consumers and producers*, which are spatial variants of the, possibly more familiar, assert and assume, respectively. To *consume/produce* an assertion is to remove/add the corresponding spatial state from/to the current symbolic state and to assert/assume the pure constraints of the assertion. Our presentation focuses on consumption, as production does not require performing frame inference, and is therefore not of immediate interest. The two main use cases for frame inference in our setting are the following:

Use of function specifications to reason about function calls. Given a function specification $\{P\}\ f(\vec{x})\ \{Q\}$, the verification tool *consumes* the part of the symbolic state that corresponds to the function pre-condition P (performing frame inference for P) and in its place *produces* a symbolic state that corresponds to the function post-condition Q.

Folding user-defined predicates. Given a predicate definition, folding a predicate consists of consuming the part of the symbolic state that corresponds to (a disjunct of) the definition and in its place producing the folded predicate, as discussed in more detail shortly.

Other use cases are similar. For example, reasoning about loops using loop invariants is largely similar to reasoning about function calls using function specifications.

2.2 Running Example: Folding a List Predicate

MPs help address two problems that arise during consumption and are related to frame inference: *the order of consumption* and *the learning of variables not given by context*, which we also refer to as *learning unknown variables*. We illustrate these problems using the example of folding the standard $\mathsf{list}(x, vs)$ predicate that describes a singly-linked list starting at address x and carrying values vs. It is defined as follows, using standard SL notation, where "$\star$" denotes the separating conjunction and "$\mapsto$" denotes the cell assertion:

$$\begin{aligned}\mathsf{list}(x, vs) \triangleq{} & (x = \mathsf{null} \star vs = [\,]) \vee \\ & (\exists v, x', vs'.\, x \mapsto v \star x+1 \mapsto x' \star \mathsf{list}(x', vs') \star vs = v : vs')\end{aligned}$$

This predicate has two disjuncts. The first disjunct states that the list is empty ($x = \mathsf{null}$) and carries no values ($vs = [\,]$). The second disjunct states that the list is non-empty, consisting of the list head node ($x \mapsto v \star x+1 \mapsto x'$), which contains the node value, v, and the pointer to the next node, x', and the tail of the list ($\mathsf{list}(x', vs')$), while connecting the values appropriately ($vs = v : vs'$, meaning that vs is the result of prepending v to vs').

Let us now attempt to fold the predicate $\mathsf{list}(x, vs)$ in the symbolic heap $\{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2, 4 \mapsto \mathsf{null}, 5 \mapsto 0, 6 \mapsto 1, 7 \mapsto 42\}$, knowing that $x = 5$.[1] As mentioned above, folding this list means performing frame inference by pinpointing a part of the symbolic state that corresponds to one of the predicate disjuncts. In consume/produce-based tools, this is done one *simple assertion* at a time, where an assertion is defined to be simple iff it does not contain the separating conjunction. Carving off the existential quantifiers, the first and second disjunct of the definition of $\mathsf{list}(x, vs)$, respectively, have the following simple assertions:

	(**B1**) $x \mapsto v$
(**A1**) $x = \mathsf{null}$	(**B2**) $x+1 \mapsto x'$
(**A2**) $vs = [\,]$	(**B3**) $\mathsf{list}(x', vs')$
	(**B4**) $vs = v : vs'$

The first disjunct is relatively straightforward: to consume it means to check if it is possible for x to equal null and for vs to equal the empty list, which it is not since we know that $x = 5$. For the second disjunct, we additionally have to *learn* the values of the existentially quantified variables v, x', and vs'. This can be more or less complex, depending on the *order* in which we process the assertions. For example, if we start with (B3), we will have to perform proof search, trying to guess the values of x' and vs' as they are not known, likely needing to backtrack and make different choices, which can be computationally expensive.

[1] For simplicity, in this example we describe symbolic heaps using cell assertions. In practice, one could choose to represent symbolic heaps differently for the purpose of, for example, efficient symbolic reasoning.

On the other hand, if we choose (B1) and (B2) first, given that we know $x = 5$, we could learn that $v = 0$ and $x' = 1$ trivially by inspecting the heap. From there, we can tackle (B3) by recursively folding the list $\mathsf{list}(x', vs')$, ultimately learning $vs' = [1, 2]$, from which we can then process (B4), learning that $vs = [0, 1, 2]$. After having folded the predicate, the remaining frame is only the single heap cell $\{7 \mapsto 42\}$.

2.3 MPs for Predicate Folding and Function Calls

MPs provide a solution to the two problems illustrated in the previous section: given an assertion P, an MP for P provides an *ordering of the simple assertions of P* so that the consumption of P is guaranteed to not backtrack, as well as *a description of how free and existentially quantified variables of P can be learnt* during this consumption.

MPs are based on dividing parameters of assertions and predicates into *input parameters* (*ins*) and *output parameters* (*outs*). Intuitively, the *ins* of an assertion/predicate are the parameters that are sufficient to be provided so that the rest of the parameters, the *outs*, can be learned. For example, for the cell assertion $x \mapsto y$, if we know x we can learn y by looking it up in the heap: therefore, the *in* of the cell assertion is x and the *out* is y. For the $\mathsf{list}(x, vs)$ predicate, on the other hand, the *in* is x and the *out* is vs.

Folding predicate example (running example). To give an example of an MP, consider again the second disjunct of the definition of the $\mathsf{list}(x, vs)$ predicate:

$$x \mapsto v \star x + 1 \mapsto x' \star \mathsf{list}(x', vs') \star vs = v : vs'$$

Assuming that only x is known before consumption, the MP for this disjunct is as follows (we give a formal definition of MPs in §4):

$$\begin{array}{l} [\ (x \mapsto v \quad\quad, [(v, \mathsf{O}_1)] \quad\quad), \\ \ \ (x + 1 \mapsto x', [(x', \mathsf{O}_1)] \quad\quad), \\ \ \ (\mathsf{list}(x', vs')\ , [(vs', \mathsf{O}_1)] \quad\ \ ), \\ \ \ (vs = v : vs', [(vs, v : vs')]\ )\ \ ] \end{array}$$

which captures the following order of the simple assertions and ways of learning variables:

1. $x \mapsto v$ comes first, and from it we learn v as the cell assertion *out* by looking up the value corresponding to address x (which we know) in the heap, which is expressed using the placeholder variable O_1;
2. $x + 1 \mapsto x'$ comes next, and from it we learn x' again as the cell assertion *out*, noting that we know the assertion *in* $x + 1$ given that we know x;
3. $\mathsf{list}(x', vs')$ comes next, and from it we learn vs' as the predicate *out*, which can be done either by recursively folding as described above, or by matching against a predicate already existing in the symbolic state; and
4. $vs = v : vs'$ comes last, and from it we learn that vs equals $v : vs'$.

Function call example. We have exemplified MPs for folding predicate definitions. MPs are equally useful to handle function specifications. Creating an MP for a function specification amounts to creating an MP for the function pre-condition, which is effectively the same as creating an MP for a disjunct of a predicate. Interestingly, the use of *ins* and *outs* has as a consequence that MPs can be created only for function pre-conditions P in which all of the variables of P can be learnt if the function parameters are known. We have observed that this is not a restriction in practice, as the parameters are the only means that a function can use to access or modify the state. In fact, a specification not obeying this property is likely either incorrect or contains resources not relevant for the function, which we can easily signal to the tool user.

2.4 MP-based Automation for Frame Inference

Gillian provides *predictable automation* for frame inference by providing machinery for automatically constructing MPs. This automation works by splitting the consumption process into two phases: a planning phase and a consumption phase – i.e., what we in the introduction of the paper referred to as "the structural and computational portions of frame inference" before having introduced consumption. An MP is automatically constructed in the planning phase. The consumption phase then follows the plan provided by the MP, which dictates consumption order and how variables are learnt. Because the planning phase is separate from the rest of consumption, planning is *predictable*. In particular, whereas the consumption phase relies on an unpredictable underlying SMT solver, the planning phase does not. The construction of MPs can therefore be understood (and, in particular, debugged) without having to take into consideration the more complicated consumption phase.

To compare Gillian with verification tools with no or little automation support for frame inference, e.g., the VeriFast tool: MPs can be said to capture the *assertion-adaptation workflow* tool users must follow when adapting assertions for such tools. Specifically, MPs capture this workflow by making clear the relationship between *ins* and *outs*. E.g., in VeriFast, the tool essentially requires that the MP can be directly "read off" assertions: the tool leaves it to the tool user to find both the consumption order and to "factor out" the *outs*, i.e., the parameters that will be learned during consumption given the *ins*. To exemplify, consider the simple assertion $x = 5$. Say x is unknown, an MP for this assertion can be directly read off the assertion: $[(x = 5, [(x, 5)])]$. Now, consider instead the simple assertion $y = x + z$ and say that y and z are known. An MP for this assertion is $[(y = x + z, [(x, y - z)])]$. In a tool without automation, the assertion would have to be adapted to $x = y - z$ such that how to learn the unknown variable x could be read off directly from the assertion. A slightly more complicated example is given by the simple assertion $x \mapsto x' + 1$ where x is known and x' is unknown. An MP for this assertion is $[(x \mapsto x' + 1, [(x', \mathsf{O}_1 - 1)])]$, meaning that to adapt the assertion to a tool without automation, a user would have to introduce an intermediate variable as follows: $x \mapsto x'' \star x' = x'' - 1$. Similarly, in tools without automation, the consumption order must be specified by the user as well. E.g., in VeriFast assertions are consumed in left-to-right order. For example, consider the (contrived but simple) assertion $x > 5 \star x = 6$. Say that x is unknown, then there is no MP where $x > 5$ is consumed first, because x cannot be learnt from $x > 5$ without guessing. Instead, the only MP for the assertion is $[(x = 6, [(x, 6)]), (x > 5, [])]$. That is, without automation support, the user would have to switch the order of the simple assertions.

Gillian automates this workflow by automatically constructing MPs, thereby automating away assertion adaptations such as the adaptations exemplified above. Moreover, the automation helps in using assertions generated by other tools (which do not necessarily generate assertions in the style verification tools expect). In §4, we provide a formal description of a simple planning algorithm that is able to construct MPs for assertions with unknown variables embedded inside simple arithmetic expressions. This simple planning algorithm is the theoretical core of the MPs and the MP-based automation of Gillian. This simple core provides a foundation that can be extended in multiple directions. We discuss some of those extensions in §5, including an example of extending the learning algorithm to handle an instance of list-based learning, which originates from a large verification case study that has been carried out in Gillian where the automation eased the amount of assertion adaptation needed.

3 Preliminaries: Assertion Language

As mentioned in the introduction, we introduce the formal description of MPs in two steps. First, in §4, we formalise a simple version of MPs, which we call core MPs. Second, in §5, we discuss extensions of core MPs that widen their applicability. In this section, we formally define the simple assertion language we use to formalise core MPs. In particular, the simple assertion language we introduce here is for the simple memory model commonly used in theoretical investigations into separation logic. When we discuss extensions of core MPs in §5, we show that core MPs are easily extended to other memory models.

Given a set of logical variables $x, y, z, \ldots \in \mathsf{LVar}$, the syntax of our assertion language is as follows:

▶ **Definition 1** (Syntax of Assertions).

$$\begin{aligned}
v \in \mathsf{Val} &\triangleq n \in \mathsf{Int} \mid b \in \mathsf{Bool} \mid \mathsf{null} \mid [\vec{v}]\\
E \in \mathsf{Exp} &\triangleq v \mid x \mid \neg E \mid E_1 \wedge E_2 \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 : E_2 \mid E_1 < E_2 \mid E_1 = E_2\\
P \in \mathsf{Asrt} &\triangleq E \mid \mathsf{emp} \mid E_1 \mapsto E_2 \mid P_1 \star P_2 \mid p(\vec{E_1}; \vec{E_2})
\end{aligned}$$

The values, Val, consist of integers, Booleans, null values, and lists of values. (Note that we will use the notation $[x]$ to denote *both* lists with elements x and the type of lists with elements of type x. E.g., $[\mathsf{LVar}]$ denotes the type of lists of LVars.) The expressions, Exp, are standard, including a representative selection of operators. We do not include program variables, as they are not needed for our discussion here; they can be treated straightforwardly. The assertions, Asrt, are also standard, except that predicate assertions, $p(\vec{E_1}; \vec{E_2})$, have their arguments separated into *ins* and *outs*, which are used to construct MPs for predicates.

Definitions of predicates (e.g., list from the overview section) come from a set Preds:

▶ **Definition 2** (Syntax of Predicates). *We describe the predicate definitions of* Preds *using the following syntax:*

$$p(\vec{x}_{in}; \vec{x}_{out}) = \bigvee_{i=1}^{n} (\exists \vec{x_i}.\ P_i)$$

where $p \in \mathsf{Str}$ *(strings),* $\vec{x}_{in}, \vec{x}_{out}, \vec{x_i} \in [\mathsf{LVar}]$*, and* $P_i \in \mathsf{Asrt}$.[2] *Predicates abide by the following restrictions:* $\vec{x}_{in}$ *and* $\vec{x}_{out}$ *have no duplicates;* $\vec{x}_{in}$ *and* $\vec{x}_{out}$ *are disjoint and for every* $i \in [1, n]$*,* $\vec{x}_{in} \cup \vec{x}_{out}$ *and* $\vec{x_i}$ *are disjoint; and* P_i *only has logical variables from* $\vec{x}_{in} \cup \vec{x}_{out} \cup \vec{x_i}$.

The semantics of assertions is standard. Let $h : \mathsf{Nat} \rightharpoonup_{fin} \mathsf{Val}$ (with $\mathsf{Nat} \subset \mathsf{Int}$) denote a heap and $\theta : \mathsf{LVar} \rightharpoonup_{fin} \mathsf{Val}$ a logical interpretation, and let $[\![E]\!]_\theta$ be the standard expression evaluation function. With these in place, the semantics of assertions is as follows:

▶ **Definition 3** (Semantics of Assertions). *The satisfaction relation for assertions, denoted by* $\theta, h \models P$*, is defined as follows:*

$$\begin{array}{lll}
\theta, h \models & E & \Leftrightarrow\ [\![E]\!]_\theta = \mathsf{true} \wedge h = \emptyset\\
& \mathsf{emp} & \Leftrightarrow\ h = \emptyset\\
& E_1 \mapsto E_2 & \Leftrightarrow\ h = \{[\![E_1]\!]_\theta \mapsto [\![E_2]\!]_\theta\}\\
& P_1 \star P_2 & \Leftrightarrow\ \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge \theta, h_1 \models P_1 \wedge \theta, h_2 \models P_2\\
& p(\vec{E_1}; \vec{E_2}) & \Leftrightarrow\ \exists i.\ \exists \vec{v_i}.\ \theta[\vec{x}_{in} \mapsto [\![\vec{E_1}]\!]_\theta, \vec{x}_{out} \mapsto [\![\vec{E_2}]\!]_\theta, \vec{x_i} \mapsto \vec{v_i}], h \models P_i\\
& & \quad \textit{for } p(\vec{x}_{in}; \vec{x}_{out}) = \bigvee_{i=1}^{n} (\exists \vec{x_i}.\ P_i) \in \mathsf{Preds}
\end{array}$$

[2] Formally, Preds is a set with elements of type $(\mathsf{Str}, [\mathsf{LVar}], [\mathsf{LVar}], [([\mathsf{LVar}], \mathsf{Asrt})])$, but this is not important for our development.

We need the following definitions in our discussion on MPs. As discussed in the overview, MPs are defined over collections of *simple assertions*:

▶ **Definition 4** (Simple Assertion). *An assertion P is* simple *iff it syntactically contains no separating conjunction: e.g., $x \mapsto 5$ is simple, and so is $\mathsf{foo}(x; y, z)$ regardless of how* foo *is defined, but $x \mapsto 0 \star y \mapsto 0$ is not simple.*

We will also need to talk about the *free logical variables of expressions and assertions*:

▶ **Definition 5** (Free Logical Variables of Expressions and Assertions). *We write $\mathtt{lv}(E)$ to denote the free logical variables of an expression E and extend this notation to lists of expressions, writing $\mathtt{lv}(\vec{E})$. The function* `lv` *naturally extends to assertions.*

4 Formalisation of Core MPs

We now formally describe a simple version of MPs, which we call core MPs, for the assertion language introduced in the previous section. We discuss extensions of core MPs in the next section.

4.1 Formal Definition of MPs

MPs are defined w.r.t. a given *knowledge base KB* and an assertion P. A knowledge base is a set of the currently known logical variables, which grows during planning. MPs have type $[(\mathsf{Asrt}, [(\mathsf{LVar}, \mathsf{Exp})])]$ and are defined as follows:

▶ **Definition 6** (Matching Plans (MPs)). *Given a knowledge base KB and an assertion P of the form $P_1 \star \cdots \star P_n$ where $P_i|_{i=1}^{n}$ are simple assertions, MP is a matching plan for P with respect to KB iff $\mathsf{plan}(KB, [P_1, \ldots, P_n], MP)$, as per the rules in Fig. 1 and Fig. 2.*

As a shorthand, we sometimes say that an assertion that has an MP is "plannable" (where the KB is usually left to be implied by context).

The rules in Fig. 1 and Fig. 2 are designed to ensure that if the simple assertions of P are consumed in the order specified by an MP of P, then the *ins* of each simple assertion P will be known before the simple assertion is consumed. To say this more formally, let us denote by $\mathit{ins}(KB, P)$ the *ins* of the assertion P under the knowledge base KB. Now, if $\mathsf{plan}(KB, [P_1, \ldots, P_n], [(P_{m_i}, \_)|_{i=1}^{n}])$ holds, then $[P_{m_i}|_{i=1}^{n}]$ is a permutation of $[P_i|_{i=1}^{n}]$, and if we let KB_i denote the knowledge base before the i-th iteration of the planning, then for every $1 \leq i \leq n$, it holds that $\mathit{ins}(KB_i, P_{m_i}) \subseteq KB_i$.

We now explain the rules in Fig. 1 and Fig. 2. We go by bottom-up order, starting with the rules for expressions.

Explanation of expression planning rules. Fig. 1 contains the rules for expression planning. The entry point into expression planning is the planExps relation. Here we discuss planning of a single expression, that is, the relation planExp, and return to the non-single case when discussing assertion planning. For core MPs, we only include a few basic learning rules for planExp to illustrate the basic idea. The rule (PURE) state that expressions where all variables are known are trivially plannable. The rule (PURE-EQ) is more interesting. It says, given an equality expression where all variables of one side are known, the expression is plannable if the unknown variables of the other side of the expression can be factored out. The factoring is done by the $\mathsf{learnEq}(KB, E_k, E_u)$ function, where E_k is known and E_u is unknown. This function, at a high level, tries to move known parts of E_u to E_k until only a

$$(\textsc{list-base})\ \frac{}{\mathsf{planExps}(KB, [\,], [\,])}$$

$$(\textsc{list-ind})\ \frac{\begin{array}{cc} 1 \leq i \leq n & \mathsf{planExp}(KB, E_i, [(x_j, E_{i_j})|_{j=1}^{k}]) \\ KB' \triangleq KB \cup \{x_j|_{j=1}^{k}\} & \mathsf{planExps}(KB', [E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n], res') \end{array}}{\mathsf{planExps}(KB, [E_1, \ldots, E_n], [(x_j, E_{i_j})|_{j=1}^{k}] \mathbin{++} res')}$$

$$(\textsc{pure})\ \frac{\mathtt{lv}(E) \subseteq KB}{\mathsf{planExp}(KB, E, [\,])} \qquad (\textsc{pure-eq})\ \frac{\begin{array}{c} \mathtt{lv}(E_i) \subseteq KB \quad \mathtt{lv}(E_j) \not\subseteq KB \\ \{i, j\} = \{1, 2\} \quad \mathsf{learnEq}(KB, E_i, E_j) = res \end{array}}{\mathsf{planExp}(KB, E_1 = E_2, res)}$$

$$\begin{aligned}
\mathsf{learnEq}(KB, E_k, x) &\triangleq [(x, E_k)] \\
\mathsf{learnEq}(KB, E_k, \neg E) &\triangleq \mathsf{learnEq}(KB, \neg E_k, E) \\
\mathsf{learnEq}(KB, E_k, E_1 + E_2) &\triangleq \begin{cases} \mathsf{learnEq}(KB, E_k - E_1, E_2), & \text{if } \mathtt{lv}(E_1) \subseteq KB, \mathtt{lv}(E_2) \not\subseteq KB \\ \mathsf{learnEq}(KB, E_k - E_2, E_1), & \text{if } \mathtt{lv}(E_1) \not\subseteq KB, \mathtt{lv}(E_2) \subseteq KB \end{cases} \\
\mathsf{learnEq}(KB, E_k, E_1 - E_2) &\triangleq \begin{cases} \mathsf{learnEq}(KB, E_1 - E_k, E_2), & \text{if } \mathtt{lv}(E_1) \subseteq KB, \mathtt{lv}(E_2) \not\subseteq KB \\ \mathsf{learnEq}(KB, E_k + E_2, E_1), & \text{if } \mathtt{lv}(E_1) \not\subseteq KB, \mathtt{lv}(E_2) \subseteq KB \end{cases}
\end{aligned}$$

Figure 1 Rules: planExps, planExp, and learnEq for expressions.

logical variable is left, which can then be learnt. The function learnEq returns a list since with support for lists in the expression language it is possible to learn multiple variables from one expression. We do not include learning rules for lists here but discuss a list-related extension in §5. Note that including learning rules for different operators is always optional: learning rules are only required to use operators in learning (i.e., to enable more automation), operators without special learning rules can still be planned as long as all unknown variables of the input assertion can be learnt elsewhere.

▶ **Example 7.** Say, $KB = \{x, y\}$. Some simple examples include

- $\mathsf{learnEq}(KB, x, z) = [(z, x)]$,
- $\mathsf{learnEq}(KB, x, z + 5) = \mathsf{learnEq}(KB, x - 5, z) = [(z, x - 5)]$,
- $\mathsf{planExp}(KB, x = z, [(z, x)])$,
- $\mathsf{planExp}(KB, x = \neg z, [(z, \neg x)])$, and
- $\mathsf{planExp}(KB, x = z + 5 - y, [(z, x + y - 5)])$.

Explanation of assertion planning rules. Fig. 2 contains the rules for assertion planning. We first discuss planSimple, which is the planning relation for simple assertions. Pure simple assertions are handled by the (CONJ) rule. The rule takes a list of expressions formed by the conjuncts of the input expression. It does so by relying on the planExps relation for planning of lists of expressions: the relation specifies expression orders for which all *outs* can be learned (its definition is similar to the definition of plan, which we discuss shortly). Non-conjunct pure simple assertions are covered by the degenerate case of the (CONJ) rule when $n = 1$. The (EMP) rule is trivial since emp is always plannable. The (HEAP) rule for cell assertions $E_1 \mapsto E_2$ states that a cell assertion is plannable if we (at least) to know its *ins*, that is, the

$$(\textsc{plan-base})\ \frac{}{\mathsf{plan}(KB, [\,], [\,])}$$

$$(\textsc{plan-ind})\ \frac{\begin{array}{ll} 1 \leq i \leq n & \mathsf{planSimple}(KB, P_i, [(x_j, E_j)|_{j=1}^{k}]) \\ KB' \triangleq KB \cup \{x_j|_{j=1}^{k}\} & \mathsf{plan}(KB', [P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n], MP) \end{array}}{\mathsf{plan}(KB, [P_1, \ldots, P_n], (P_i, [(x_j, E_j)|_{j=1}^{k}]) : MP)}$$

$$(\textsc{conj})\ \frac{n \geq 1 \qquad \mathsf{planExps}(KB, [E_1, \ldots, E_n], \mathit{res})}{\mathsf{planSimple}(KB, \wedge_{i=1}^{n} E_i, \mathit{res})}$$

$$(\textsc{emp})\ \frac{}{\mathsf{planSimple}(KB, \mathsf{emp}, [\,])} \qquad (\textsc{heap})\ \frac{\begin{array}{c} \mathtt{lv}(E_1) \subseteq KB \\ \mathsf{planExps}(KB \cup \{\mathsf{O}_1\}, [E_2 = \mathsf{O}_1], \mathit{res}) \end{array}}{\mathsf{planSimple}(KB, E_1 \mapsto E_2, \mathit{res})}$$

$$(\textsc{pred})\ \frac{\begin{array}{c} \mathtt{lv}(\overrightarrow{E_1}) \subseteq KB \qquad \overrightarrow{E}_2 = [E_{2_i}|_{i=1}^{n}] \\ \mathsf{planExps}(KB \cup \{\mathsf{O}_1, \ldots, \mathsf{O}_n\}, [E_{2_1} = \mathsf{O}_1, \ldots, E_{2_n} = \mathsf{O}_n], \mathit{res}) \end{array}}{\mathsf{planSimple}(KB, p(\overrightarrow{E_1}; \overrightarrow{E_2}), \mathit{res})}$$

Figure 2 Rules: plan and planSimple for assertions.

logical variables of E_1, and E_2 is plannable according to planExps. We also need to record that the *out* of the cell assertion, that is, E_2, equals the contents of the cell at E_1 in memory, and from this equality we may be able to learn further information. Note, however, that the heap is not available during the planning process, and we therefore use a placeholder variable O_1, which is a reserved logical variable that is not allowed to occur in assertions, that will be instantiated to the actual heap contents at runtime (see Ex. 8). Finally, the (Pred) rule generalises the planning of cell assertions to predicates by abstracting all of the predicate *outs* using placeholder variables $\mathsf{O}_1, \ldots, \mathsf{O}_n$ to then be instantiated and linked to $E_{2_i}|_{i=1}^{n}$ appropriately.

▶ **Example 8.** To illustrate how placeholder variables work in practice, consider consuming $x \mapsto y$ knowing that $x = 41$. An MP for this assertion, by (heap), is $[(x \mapsto y, [(y, \mathsf{O}_1)])]$. Say the current heap has a cell $41 \mapsto 42$. In this case, at the time of consumption the placeholder variable O_1 will be instantiated with the contents of the cell at x, which equals 42, yielding $y = 42$.

We now turn to the main entry point: the plan relation. The main rule of plan, the (plan-ind) rule, specifies valid orders of the simple assertions of P that guarantee the learning of all their *outs*, extending the knowledge base as the *outs* of each simple assertion are learnt. At each choice point, the rule is applicable for a simple assertion P_i whose *ins* are all known using the planSimple relation, together with all the logical variables $x_j|_{j=1}^{k}$ that can be learnt from P_i and expressions $E_j|_{j=1}^{k}$ describing how they can be learnt. The rule then extends the knowledge base with learnt variables, inductively repeats the planning for the remaining simple assertions, and finally adds the result of planSimple, $(P_i, [(x_j, E_j)|_{j=1}^{k}])$, to the full MP.

▶ **Example 9.** Let $P = \mathsf{list}(w; vs) \star x \mapsto y \star z \mapsto w \star z = y + 21$ and $KB = \{x\}$, and let us use the provided rules to construct an MP for P. Branching on the (plan-ind) rule, we have that for $i = 1$ and $i = 3$ we cannot apply (pred) and (heap) as we do not know all the corresponding ins (in particular, w and z), and for $i = 4$ we cannot apply (PURE-EQ) as we do not know all of the variables on either side of the equality. For $i = 2$, however, we can apply (HEAP) as we know x and we have $\mathsf{planExps}(KB \cup \{\mathsf{O}_1\}, [y = \mathsf{O}_1], [(y, \mathsf{O}_1)])$, which follows from $\mathsf{planExp}(KB \cup \{\mathsf{O}_1\}, y = \mathsf{O}_1, [(y, \mathsf{O}_1)])$, which in turn follows from $\mathsf{learnEq}(KB \cup \{\mathsf{O}_1\}, \mathsf{O}_1, y) = [(y, \mathsf{O}_1)]$. Returning to (PLAN-IND) and continuing until the end, we obtain the following MP for P:

$$[(x \mapsto y, [(y, \mathsf{O}_1)]), (z = y + 21, [(z, y + 21)]), (z \mapsto w, [(w, \mathsf{O}_1)]), (\mathsf{list}(w; vs), [(vs, \mathsf{O}_1)])]$$

4.2 Computing MPs

Given the inference-rule formalisation of MPs in the previous section, it is easy to construct an algorithm for automatically constructing MPs: a simple greedy algorithm searching through the plan relation is sufficient to find an MP for a given assertion. We can greedily pick the first valid choice we find at each choice point of the rules of the plan relation and its auxiliary relations. That is, no backtracking is needed to explore multiple choice points (note that here we are referring to backtracking during the construction of MPs, not the backtracking during consumption that MPs help to avoid as discussed earlier). This is because *outs* are only learnt by equality reasoning and therefore learning only happens when forced, so the order in which *outs* are learnt does not matter. In §6, we report performance numbers of this simple greedy algorithm as implemented in Gillian.

Note that no soundness result is needed for MPs to ensure soundness for the verification tool as a whole: as long as no simple assertions are dropped from a given input assertion, it is not possible to construct an "incorrect" MP that leads to an unsoundness bug in the verification tool. This is because an incorrectly constructed MP will simply make the consumption following the MP to fail and force the verification process to abort. To exemplify, consider the assertion $x = 1$ with an empty knowledge base. Say we construct the incorrect MP $[(x = 1, [(x, 0)])]$, suggesting to instantiate x to 0. During consumption, this MP will lead to $0 = 1$ being consumed, which will of course fail. Similarly, an MP missing one or more *outs* will cause the consumption to fail as well. E.g., an incorrect MP $[(x = 1, [])]$ for the same assertion, where the x variable is missing, will be caught during consumption as well since x will be left uninstantiated.

4.3 MPs for Function Specifications and Predicates

Given the definition of an MP for an assertion, we can easily define MPs for function specifications and predicates, as we now explain and exemplify.

MPs for function specifications are defined as follows:

▶ **Definition 10** (Matching Plans: Function Specification). *An MP for a function specification* $\{\vec{\mathtt{x}} = \vec{x} \star P\}\, f(\vec{\mathtt{x}})\, \{Q\}$ *is an MP for* P *with knowledge base* $\{\vec{x}\}$*.*

For function specifications of the above form, where the function parameters $\vec{\mathtt{x}}$ are bound to logical variables $\vec{x}$, when symbolically executing a function call, the values of $\vec{x}$ are given by the arguments provided in the call, and $\vec{x}$ can therefore be assumed to be known at the start of the planning.[3]

MPs for predicates are defined as follows:

[3] As we do not include program variables in assertions, pre-conditions are formally pairs of the form $(\vec{x}, P)$, but we stylise them to remain in line with the usual SL syntax.

▶ **Definition 11** (Matching Plans: Predicates). *An MP for a predicate $p(\vec{x}_{in}; \vec{x}_{out}) = \bigvee_{i=1}^{n}(\exists \vec{x_i}.\ P_i)$, is a list of MPs, $[mp_i|_{i=1}^{n}]$, such that mp_i is an MP for P_i with knowledge base $\{\vec{x}_{in}\}$.*

Recall that the use case for MPs for predicates is predicate folding: to fold a predicate $p(\vec{x}_{\text{in}}; \vec{x}_{\text{out}})$ we have to know its *ins* $\vec{x}_{\text{in}}$, whereas the existentials from the predicate body disjuncts need to be inferable from these *ins* and the *outs* $\vec{x}_{\text{out}}$ can be either provided or optionally left to be inferred from the *ins*. Also note that how MPs are defined for predicates does not depend on how much folding automation the verification tool provides: from the perspective of planning, it does not matter if the fold was requested manually or automatically. Lastly note that because the *ins* and *outs* of a predicate are given at the time of definition, failure to construct an MP can be reported early, i.e., at the time of definition, rather than when the predicate is used in folding.

Examples of plannable predicates include all predicates for standard data structures. We discuss some data-structure examples in more detail below.

▶ **Example 12.** We return to the standard SL predicate $\mathsf{list}(x; vs)$ from our running example, where we now have separated the *ins* and *outs*. Recall, the predicate is defined as follows:

$$\begin{aligned}\mathsf{list}(x; vs) \triangleq\ & (x = \mathsf{null} \star vs = [\,]) \lor \\ & (\exists v, x', vs'.\ \mathsf{list}(x'; vs') \star x \mapsto v \star x + 1 \mapsto x' \star vs = v : vs')\end{aligned}$$

Importantly, despite the fact that the definition of the list predicate is recursive, no recursion is needed to express (or compute) the MP for the predicate. Per Def. 11, the predicate is plannable since the following is an MP for the predicate:

$$\begin{array}{ll}[[(x = \mathsf{null}, & [(x, \mathsf{null})]), \\ \ (vs = [\,], & [(vs, [\,])])], \\ \ [(x \mapsto v, & [(v, \mathsf{O}_1)]), \\ \ \ (x + 1 \mapsto x', & [(x', \mathsf{O}_1)]), \\ \ \ (\mathsf{list}(x'; vs'), & [(vs', \mathsf{O}_1)]), \\ \ \ (vs = v : vs', & [(vs, v : vs')])]]\end{array}$$

where the first element of the list is an MP for the first disjunct of the predicate body (the null disjunct) and the second element of the list is an MP for the second disjunct of the predicate body (the non-null disjunct).

▶ **Example 13.** We easily see that the following two variants of the singly-linked list predicate list and the doubly-linked list predicate dlist are plannable:

$$\begin{aligned}\mathsf{list}(x) &\triangleq (x = \mathsf{null}) \lor (\exists v, x'.\ x \mapsto v, x' \star \mathsf{list}(x')) \\ \mathsf{list}(x; n) &\triangleq (x = \mathsf{null} \star n = 0) \lor (\exists v, x'.\ x \mapsto v, x' \star \mathsf{list}(x'; n - 1)) \\ \mathsf{dlseg}(x, x', y, y'; vs) &\triangleq (vs = [\,] \star x = x' \star y = y') \lor \\ & \quad\ (\exists x'', v, vs'.\ vs = v : vs' \star x \mapsto v, x'', y' \star \mathsf{dlseg}(x'', x', y, x; vs')) \\ \mathsf{dlist}(x, y; vs) &\triangleq \mathsf{dlseg}(x, \mathsf{null}, y, \mathsf{null}; vs)\end{aligned}$$

▶ **Example 14.** For a non-list example of a plannable data-structure predicate, we turn to binary search trees (extending our simple assertion language's values and expressions with support for sets):

$$\begin{aligned}\mathsf{bst}(x; vs) \triangleq\ & (x = \mathsf{null} \star vs = \emptyset) \lor \\ & (\exists v, l, r, vs_l, vs_r.\ x \mapsto v, l, r \star \mathsf{bst}(r; vs_r) \star \mathsf{bst}(l; vs_l) \star \\ & \ vs = vs_l \uplus \{v\} \uplus vs_r \star vs_l < v \star v < vs_r)\end{aligned}$$

▶ **Example 15.** Finally, we highlight that is up to the tool user to make sensible choices for *ins* and *outs*, as not all plannable choices need be equally useful in practice. Note that this is not a requirement introduced by MPs, rather, MPs simply make this requirement explicit by separating *ins* from *outs*. To illustrate, consider the standard acyclic- and cyclic-list-segment predicates:

$$\begin{aligned}\mathsf{lseg}(x, y, vs) &\triangleq (x = y \star vs = [\,]) \vee \\ &\quad (\exists x', v, vs'.\, x \neq y \star x \mapsto v, x' \star vs = v : vs' \star \mathsf{lseg}(x', y, vs')) \\ \mathsf{clseg}(x, y, vs) &\triangleq (x = y \star vs = [\,]) \vee \\ &\quad (\exists x', v, vs'.\, x \mapsto v, x' \star vs = v : vs' \star \mathsf{clseg}(x', y, vs'))\end{aligned}$$

where the only difference between the two is in that the former does not allow the start and the end pointers to be equal in the second disjunct of its definition (specified by $x \neq y$) and the latter does not have this constraint, and consider the various choices of *ins* and *outs*, with the goal being that the *ins* should uniquely determine the *outs*, minimising potential branching coming from folding. For both lseg and clseg, x has to be an *in*, as otherwise, given that the list is singly-linked (forward-pointing), we would have no way of determining where the list segment starts. Observe that only having x as an *in* is enough for both disjuncts in both predicate definitions to be plannable. However, without additional *ins*, we do not know where the list segment ends, and folding the predicate would yield up to n branches, where n is the length of the maximal list segment in the heap starting from x. Adding y as an *in* solves this issue for lseg, since then we fix the list segment by knowing both the start and its end; similarly, we could add vs as an *in* and then we would know the length of the list segment, which, together with its start, would also uniquely determine it. Interestingly, adding y as an *in* for clseg still does not uniquely determine the cyclic list segment, as its two disjuncts are not disjoint: for example, in the heap $\{42 \mapsto 0, 42\}$, we could fold both $\mathsf{clseg}(42, 42; [\,])$ and $\mathsf{clseg}(42, 42; [0])$. Adding vs as an *in* of clseg, however, does solve the issue, as the length of the list segment then becomes unambiguous. The same situation would come up with any predicate whose disjuncts are not disjoint.

5 Extensions

Having formalised core MPs in the previous section, we now discuss important MP extensions that widen the applicability and usefulness of MPs. The extensions we discuss here are from the implementation of MPs in the Gillian tool.

Parametric matching plans. To support multiple programming languages (e.g., C and JavaScript), Gillian is parametric on the memory model used for analysis. In supporting parametricity, Gillian's implementation of MPs is parametric as well, which we now show is a simple extension of core MPs.

Memory models in Gillian are described in terms of *core predicates*, which represent the fundamental units of the memory model. Core predicates are described using core-predicate assertions with syntax $c(\overrightarrow{E_1}; \overrightarrow{E_2})$, where $c \in \mathsf{Str}$ is the name of the core predicate and $\overrightarrow{E_1}$ and $\overrightarrow{E_2}$ are the *ins* and *outs* of the core predicate. Each memory model instance must provide a set of core predicates and the *ins* and *outs* of each core predicate. For example, for the simple memory model we used for core MPs, the only core predicate is the cell assertion, $E_1 \mapsto E_2$, which has E_1 as an *in* and E_2 as an *out* – or, more formally: $\mapsto(E_1; E_2)$. Another example is given by the Gillian C memory model, whose core predicates include a cell core predicate of the form $(E_b, E_o) \mapsto E_v$, which states that the cell at offset E_o in the block at location E_b has contents E_v, where E_b and E_o are the *ins* and E_v is an *out*, and a block-bound predicate $\mathsf{bound}(E_b; n)$, which states that the block at location E_b has length n.

From the discussion above, it is straightforward to generalise planning to parametric planning since core-predicate assertions $c(\overrightarrow{E_1}; \overrightarrow{E_2})$ share syntax with user-defined-predicate assertions $p(\overrightarrow{E_1}; \overrightarrow{E_2})$ and therefore for the purpose of planning are the same. That is, the (PRED) rule of Fig. 2 can be used to plan core-predicate assertions. Indeed, recall that for the simple memory model we used for core MPs, the (HEAP) rule is indeed a special case of the (PRED) rule (see Fig. 2).

Extending learning capabilities. In some large verification projects, it might be desirable to extend the learning capabilities of the core MP algorithm with project-custom learning rules: for example, to avoid repetitive manual project-specific massage of assertions to make them plannable with respect to the simple learning rules of core MPs.

We discuss one such learning extension that has been implemented in Gillian, specifically, a list-related extension that was added for the largest case study carried out in Gillian: the verification of C and JavaScript implementations of the deserialisation module of the AWS Encryption SDK message header [13]. To illustrate, consider the assertion $P \triangleq a = a_l \mathbin{++} a_r \star \mathsf{len}(a_l) = l$ with $KB = \{a, l\}$, where $++$ denotes list concatenation and len denotes list length. P is not plannable using the core MP algorithm, because the algorithm can only learn logical variables: as list length is not injective, a_l cannot be learned from l and planning is stuck. However, P becomes plannable if knowledge bases are allowed to also contain expressions of the form $\mathsf{len}(x)$: $\mathsf{len}(a_l)$ can then be learnt from $\mathsf{len}(a_l) = l$, and both a_l and a_r can be learnt, respectively, as $a_l = a[0 : \mathsf{len}(a_l)]$ and $a_r = a[\mathsf{len}(a_l) : \mathsf{len}(a)]$ from $a = a_l \mathbin{++} a_r$, where $E_1[E_2 : E_3]$ denotes list slicing from index E_2 inclusive to index E_3 exclusive.

The above example may look simple but was essential for creating MPs of predicates describing the data structures used in the AWS case study. At a high level, AWS Encryption SDK message headers are buffers (arrays of bytes) that comprise a number of sections, with each section having either a static length described by the standard or a dynamic length derived from content appearing in the earlier sections of the buffer. In that context, the list-length extension allowed for clear definitions that follow the descriptions in their official documentation. Otherwise, the predicates would have to be stated using more complex operators. Specifically, using Gillian notation, (part of) the predicate describing the message header is as follows:[4]

```
pred Header(+rawHeader, ver, type, sId, msgId, ECLen, ECKs, ...) :
  rawHeader == ([ ver, type ] ++ #rawSId ++ msgId ++ #rawECLen ++ #EC ++ ...) *
  len(#rawSId) == 2   * UInt16(#rawSId, suiteId) * len(msgId) == 16 *
  len(#rawECLen) == 2 * UInt16(#rawECLen, ECLen) *
  len(#EC) == ECLen   * EncryptionContext(#EC, ECKs) * ...
```

while without the list-length extension its definition would be as follows:

```
pred Header(+rawHeader, ver, type, sId, msgId, ECLen, ECKs, ...) :
  [ ver, type ] == rawHeader[0, 2] * #rawSId == rawHeader[2, 4] *
  UInt16(#rawSId, suiteId) * msgId == rawHeader[4, 16] *
  #rawECLen = rawHeader[20, 22] * UInt16(#rawECLen, ECLen) *
  #EC == rawHeader[22, 22 + ECLen] * EncryptionContext(#EC, ECKs) * ...
```

[4] In Gillian notation, the `+` symbol denotes a predicate *in*, and the `#` symbol denotes existential quantification. The `UInt16(+x, y)` predicate states that the two bytes given by `x` can be viewed as an unsigned 16-bit integer `y`, while the `EncryptionContext` predicates is specific to the AWS case study and its meaning is not relevant here.

By comparing these two definitions, we can see that not only is the latter more difficult to read and understand, but is also more error-prone, as the list-slicing indices get progressively more complicated.

This extension approach is not limited to the above list-length example and can be applied for other expressions: e.g., we might choose to keep $a + b$ in the KB if we know it but do not know either a or b. These further extensions can be added on an as-needed basis straightforwardly by modifying the OCaml code of Gillian. An interesting direction for the future is to develop a small domain-specific language for MP rules (i.e., rules like those in e.g. Fig. 1) to simplify extending Gillian's MP algorithm with new rules for extended learning capabilities.

Support for magic wands. MPs can easily be extended to support the magic wand operator $-\!\!\ast$. Formally, $\theta, h \models P_1 -\!\!\ast P_2 \Leftrightarrow (\forall h'.\ h' \# h \wedge \theta, h' \models P_1 \Rightarrow \theta, (h' \uplus h) \models P_2)$ where $h' \# h$ denotes that the heaps are disjoint. Practically, magic wands are helpful to reason about "the rest" of a structure. For example, iterating over a linked list often requires the introduction of the list-segment predicate `lseg`, presented in Ex. 15, in order to specify the beginning of the list that has already been visited. Instead, the list segment $\texttt{lseg}(x, y, vs)$ can be replaced by the magic wand $\texttt{list}(y, vs') -\!\!\ast \texttt{list}(x, vs \mathbin{++} vs')$, meaning that the total list can be recovered by combining this resource with the rest of the list.

To add support for magic wands, we extend the assertion language with magic wand assertions of the form $p(\overrightarrow{E_1}; \overrightarrow{E_2}) -\!\!\ast q(\overrightarrow{E_3}; \overrightarrow{E_4})$, where p and q are user-defined predicates. We chose this syntax as to syntactically capture the in-parameters and out-parameters for each side of the operator. Such a magic wand assertion forms a simple assertion with $\overrightarrow{E_1}$, $\overrightarrow{E_2}$ and $\overrightarrow{E_3}$ as in-parameters, and $\overrightarrow{E_4}$ as outs-parameters. To explain the division of in-parameters and out-parameters, we give a summary of the underlying algorithm for performing consumption of magic wands as implemented in Gillian. The algorithm is originally from Viper [23, 5]:[5]

```
To consume a magic wand assertion $p(\overrightarrow{E_1}; \overrightarrow{E_2}) -\!\!\ast q(\overrightarrow{E_3}; \overrightarrow{E_4})$ from state $\sigma$:
1. Create a state $\sigma_p$ by producing the definition of $p(\overrightarrow{E_1}; \overrightarrow{E_2})$ in the empty state.
2. For each simple assertion $Q$ in the definition of $q(\overrightarrow{E_3}; \overrightarrow{E_4})$:
    a. Try consuming $Q$ in $\sigma_p$, if it succeeds continue to the next simple assertion;
    b. If it fails, try consuming $Q$ in $\sigma$ instead, if it succeeds, continue to the next simple
       assertion;
    c. If both fail, abort the consumption.
```

Step 1 produces the left-hand side of wand, which requires knowing all its parameters. Therefore, all parameters in $\overrightarrow{E_1}$ and $\overrightarrow{E_2}$ must be in-parameters of the wand assertion. Then, step 2 consumes the right-hand side of the wand, which requires knowing all its in-parameters, but learns its out-parameters in the process. Therefore, $\overrightarrow{E_3}$ are in-parameters of the wand assertion and $\overrightarrow{E_4}$ are out-parameters.

To exemplify, say $p_1(x; y) = x \mapsto y$ and $q_1(x; y, z) = x \mapsto y \star y \mapsto z$. Further, say we know $x = 1$ and $y = 2$ and are in a state with a heap $\{2 \mapsto 3\}$. The heap satisfies the wand assertion $\exists z.\ p_1(x; y) -\!\!\ast q_1(x; y, z)$. Indeed, by the above algorithm, the assertion can be consumed starting from the given heap, learning that $z = 3$ in the process.

[5] For simplicity of presentation, the algorithm presented here assumes the absence of disjunction in the definitions of p and q.

6 Scalability and Performance

We now discuss the scalability and performance of MPs. We base our discussion on the MP implementation in Gillian, specifically, our discussion builds on the largest case study carried out in Gillian, i.e., the verification of C and JavaScript implementations of the deserialisation module of the AWS Encryption SDK message header [13]. First, we report MP-related scale and performance data for the AWS case study. Second, we report on a new MP-based optimisation we have implemented in Gillian for this paper, which allows for the creation of *aggregate matching plans* (AMPs). We show that this optimisation improves the total verification time of the AWS case study.

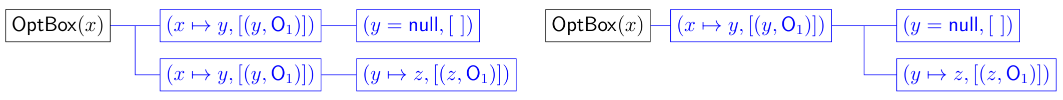
AWS case study. To measure the scale and performance of MPs, we have instrumented Gillian with data-and-performance counters and re-run the verification of the code from the AWS case study. From this experiment, we have found the cost of building MPs to be negligible compared to the total verification time. For the C/JS implementation of the AWS case study, building all MPs takes a total of 0.35s/0.096s, constituting 0.16%/0.25% of the total verification time. Over that time, MPs are built for 1073/378 assertions that consist of 41/28 simple assertions on average and 156/272 assertions at most. The creation of a single MP takes 0.33ms/0.26ms on average and 2.5ms/6.5ms at most. Note that MPs do not affect the verification time beyond the time it takes to create them; this is because MPs are separated from the consumption phase: the consumptions that take place during verification would be the same if the input assertions had instead been manually adopted (e.g., as illustrated in the discussion on VeriFast in §2).

Aggregate matching plans (AMPs). We discuss and evaluate *aggregate matching plans* (AMPs), a new performance optimisation we have implemented in Gillian for this paper. To illustrate AMPs, recall that an MP for a predicate is a list of MPs for the disjuncts of the body of the predicate (Def. 11), and that each disjunct is treated independently. AMPs identify and leverage simple assertions that are shared between disjuncts and represent this sharing within a tree structure.

To better understand how AMPs work, consider the following predicate:

$$\mathsf{OptBox}(x) \triangleq (\exists y.\ x \mapsto y \star y = \mathsf{null}) \vee (\exists y, z.\ x \mapsto y \star y \mapsto z)$$

and the MPs and AMP for this predicate in Fig. 3.



$\mathsf{OptBox}(x)$
$(x \mapsto y, [(y, \mathsf{O}_1)])$
$(y = \mathsf{null}, [\,])$
$(x \mapsto y, [(y, \mathsf{O}_1)])$
$(y \mapsto z, [(z, \mathsf{O}_1)])$
$\mathsf{OptBox}(x)$
$(x \mapsto y, [(y, \mathsf{O}_1)])$
$(y = \mathsf{null}, [\,])$
$(y \mapsto z, [(z, \mathsf{O}_1)])$

(a) The matching plans. **(b)** The aggregate matching plan.

Figure 3 Matching plans and aggregate matching plan for $\mathsf{OptBox}(x)$.

Without AMPs, Gillian would create 2 MPs, one per disjunct of the predicate, which both have the same first step $(x \mapsto y, [(y, \mathsf{O}_1)])$. However, when folding a predicate, Gillian tries to consume each disjunct of the predicate body in order until one succeeds to completion. Without AMPs, when the definition that could be folded was the second one, the first step would be consumed twice in the same symbolic state, duplicating the work. In contrast, using AMPs it is consumed only once, factoring out such duplicated work.

In our implementation of AMPs in Gillian, MPs are built for each disjunct of a predicate, and then aggregated into a single AMP. Before building the individual MPs, simple assertions within a single disjunct are sorted using a simple sort algorithm, maximising the chance of the existence of a shared root. A similar process is also performed for function specifications, as each function can have multiple specifications in Gillian.

Our evaluation of this new optimisation shows that utilising AMPs instead of lists of MPs in large verification projects leads to substantial performance improvements. For the C implementation of the AWS case study, AMPs made the total verification time drop from 240s to 211s, that is, a speedup of 12%. AMPs are especially effective when assertions are obtained from and/or augmented by a compilation process which often adds the same contextual information, such as type information, to all cases (which is the case for the Gillian assertion compiler for C).

7 Related Work

We place matching plans in the context of previous work on automated frame inference. Specifically, we compare matching plans with the approaches of three modern SL-adjacent and SMT-based semi-automated verification tools: VeriFast [7], Viper [16], and CN [20].

VeriFast. VeriFast [7], whose approach is closest to our work, is a verification tool for C and Java. It is based on consumers and producers, and its assertion language is the traditional SL assertion language. Given the similarities between VeriFast and our work, in particular the shared assertion language, we expect it would be straightforward to adapt our work on MPs for VeriFast. Currently, the approach of VeriFast offers less automation than MPs, as it leaves the responsibility of constructing MPs to the tool user, who has to provide the MP implicitly when providing assertions, e.g., as part of predicate definitions. To illustrate, consider again the singly-linked list predicate from our running example, now in VeriFast's syntax for C:

```
struct node { int entry; struct node* next; };

predicate list(struct node* x, list<int> vs) =
  x == NULL
  ? vs == nil
  : malloc_block_node(x) &*& x->entry |-> ?v &*&
    x->next |-> ?x’ &*& list(x’, ?vs’) &*& vs == cons(v, vs’);
```

Note that this list predicate is defined using the ternary conditional operator rather than disjunction, and that existentially quantified variables are annotated with a question mark at first use. In VeriFast, simple assertions are consumed in the order given by the tool user: e.g., in the non-`NULL` case of the list predicate, `malloc_block_node(x)` is consumed before `x->entry |-> ?v`, which in turn is consumed before `x->next |-> ?x’`, and so on. This means that if the user does not arrange the simple assertions appropriately, the verification will fail even though there might exist an MP. Further, VeriFast offers less automation than MPs w.r.t. learning variables: it can only learn a variable if that variable is the single occupant of an *out* or the left-hand side of an equality: for example, assuming `x` is known, VeriFast can learn `y` from `y == x` but not from `x == y` or `x == y + 1`.

Another difference between our approach and that of VeriFast is that in our approach *ins* and *outs* are checked at definition time whereas in VeriFast they are checked at use time, leading to less local/precise error reporting. For example, if we tried to fold a list in VeriFast using `close list(_, _)`, where `_` denotes that VeriFast should infer the argument, we would get the error message "Unbound variable 'x' ", referring to the `x` variable in the `list` predicate definition, instead of an error saying that it is not possible to infer an *in* of a predicate.[6]

Viper. Viper [16] is a platform for building verification tools. It has been instantiated, among other languages, to Java and Rust. It is based on consumers and producers, but also on an alternative assertion language known as implicit dynamic frame theory (IDF) [24], which combines SL with dynamic frame theory [10].[7] Tool users familiar with SL but not IDF must therefore learn IDF before they can start using Viper. This difference also means that both consumption and learning *outs* from *ins* look different than in our setting, making a detailed comparison complex. We illustrate this using our linked-list running example, now in Viper's IDF syntax:

```
field entry : Int
field next  : Ref

predicate list(this : Ref) {
  acc(this.entry) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}

function elems(this : Ref) : Seq[Int]
requires list(this) {
  unfolding list(this) in
  this.next == null ? Seq(this.entry)
                    : Seq(this.entry) ++ elems(this.next)
}
```

The above `list` predicate captures the shape, but not the contents, of lists. The predicate is expressed using `acc`, a construct called accessibility predicate, closely resembling the cell assertions in SL. The contents of lists are specified using a heap-dependent function `elems`. Such functions, as their name suggests, are functions over the heap of the current symbolic state. In the setting of accessibility predicates and heap-dependent functions, the *ins* look similar to *ins* in our setting, but the *outs* become the return values of heap-dependent functions. Assertions must be self-framing, in the sense that assertions must ensure accessibility to at least the locations they read. Self-framedness is checked in a left-to-right manner in Viper, meaning that the assertion `acc(x.f) && 0 < x.f` is considered self-framing, whereas `0 < x.f && acc(x.f)` is not. That is, like VeriFast, Viper is sensitive to the order of simple assertions. Quantifiers, e.g., over array indices, are more prominent in IDF than in SL, and variables, quantified or otherwise, that cannot be inferred are instantiated by giving the underlying SMT solver trigger hints [14], in the style of, e.g., Boogie [12].

CN. CN [20] is a verification tool for C, and is designed for, what its authors call, "predictable proof automation". One means employed towards this goal is that CN is based on a new tool-specific assertion language, which uses variable scoping to ensure that *outs* can always be learnt. In other words, the limitations of CN's learning algorithm are reflected directly in

[6] VeriFast has support for checking "preciseness of predicates", which allows for definition-time checking of their *ins* and *outs*. However, this feature does not affect the error reporting at use sites of predicates, i.e., errors remain nonlocal. The rules are the same as for the run-time check and are described by inference rules by Jacobs et al. [7] and by prose text by Jacobs et al. [8].

[7] Parkinson and Summers [19] establish a formal connection between SL and IDF, and Jost and Summers [9] (partially) extend the result to include predicates as well.

syntax of assertions, ensuring that tool users do not accidentally fall out of the plannable subset of the assertion language. To exemplify, in CN syntax, the singly-linked list predicate for the same `node` data type as in the above discussion on VeriFast becomes:[8]

```
predicate { list<integer> l } List (pointer p) {
  if (p == NULL) {
    return { l = nil<integer> };
  } else {
    let Head = Owned<struct node>(p);
    let Tail = List(Head.value.next);
    return { l = cons(Head.value.entry, Tail.l) };
  }
}
```

Since new variables, including *outs*, must be the output of functions, they are necessarily learnable. The downside of this approach is that tool users have to learn a new specification language.

Another feature aimed at predictable proof automation is that CN targets a decidable SMT fragment, disallowing, e.g., nonlinear arithmetic in SMT queries. Instead, these must be handled manually by tool users, by proving lemmas in, e.g., Coq, and then manually applying them in CN. This trade-off is not CN-specific and could also be done in an MP-based approach. Similarly, manual fallbacks for complex quantifiers are required as well.

Other related work. Many other SL and SL-adjacent verification tools share similarities with the work presented here, all the way back from Smallfoot [2, 3], the very first such tool. Important differences between our work and Smallfoot include that Smallfoot is not SMT-based and that its frame inference procedure is more akin to proof search than the approach presented here, as is the case for its most well-known descendant Infer [4]. Another important approach to semi-automating SL is embedding one's verification tool inside an interactive theorem prover (ITP). A recent example of this approach is RefinedC [22]. Such tools do not reduce the verification problem to a series of SMT queries but instead to a series of proof obligations that tool users must then discharge within the ITP by the usual means available, including various proof automation machinery.

There are also important connections to be highlighted between the work presented here and logic programming. The above-mentioned work on RefinedC [22] highlights this connection, as the tool is implemented in the "separation logic programming language" Lithium (which, in turn, is implemented in Coq), which is introduced in the same paper. Diaframe [15] is based on similar ideas. Nguyen et al. [17] highlight the connection between what we call *ins* and *outs* and argument modes in logic programming. Lastly, some logic programming languages contain features that address some of the problems of the traditional left-to-right evaluation order of logic programming, such as constraint logic programming and co-routining (e.g., `dif/2`) (cf. the recent survey by Körner et al. [11]).

Finally, an earlier version of MPs, dubbed unification plans (UPs), was briefly outlined by Fragoso Santos et al. [6] in the context of the JavaScript analysis tool JaVerT, the forefather of the Gillian platform. UPs featured a more limited form of learning than our MPs and were constructed purely syntactically: *ins* and *outs* were computed independently of a knowledge base, which meant that, for example, while it was possible to learn b from $b = a$ or $\mathsf{list}(a; b)$ knowing a, it was not possible to learn c from $a = b + c$ or learn list lengths. No paper has covered UPs in the same depth we have covered MPs in this paper.

[8] The definition is taken from the CN paper, where the authors add: "Note that CN does not currently support logical functions on lists; this example is for illustration only."

References

1 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018. doi:10.1145/3182657.

2 J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Conference on Formal Methods for Components and Objects*, 2005. doi:10.1007/11804192_6.

3 J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Asian Conference on Programming Languages and Systems*, 2005. doi:10.1007/11575467_5.

4 Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_33.

5 Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. Sound automation of magic wands. In *Computer Aided Verification*, 2022. doi:10.1007/978-3-031-13188-2_7.

6 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290379.

7 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_4.

8 Bart Jacobs, Jan Smans, and Frank Piessens. *The VeriFast Program Verifier: A Tutorial*, 2017. doi:10.5281/ZENODO.1068185.

9 Daniel Jost and Alexander J. Summers. An automatic encoding from VeriFast predicates into implicit dynamic frames. In *Verified Software: Theories, Tools, Experiments*, 2014. doi:10.1007/978-3-642-54108-7_11.

10 Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Symposium on Formal Methods*, 2006. doi:10.1007/11813040_19.

11 Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. Fifty years of Prolog and beyond. *Theory and Practice of Logic Programming*, 22(6), 2022. doi:10.1017/S1471068422000102.

12 K. Rustan M. Leino. This is Boogie 2, 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

13 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: Real-world verification for JavaScript and C. In *Computer Aided Verification*, 2021. doi:10.1007/978-3-030-81688-9_38.

14 Michał Moskal. Programming with triggers. In *Workshop on Satisfiability Modulo Theories*, 2009. doi:10.1145/1670412.1670416.

15 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *Conference on Programming Language Design and Implementation*, 2022. doi:10.1145/3519939.3523432.

16 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, 2016. doi:10.1007/978-3-662-49122-5_2.

17 Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *Verification, Model Checking, and Abstract Interpretation*, 2008. doi:10.1007/978-3-540-78163-9_19.

18 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, 2001. doi:10.1007/3-540-44802-0_1.

19 Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *European Symposium on Programming*, 2011. doi:10.1007/978-3-642-19718-5_23.

20 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: Verifying systems C code with separation-logic refinement types. *Proceedings of the ACM on Programming Languages*, 7(POPL), 2023. doi:10.1145/3571194.

21 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, 2002. doi:10.1109/LICS.2002.1029817.

22 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *International Conference on Programming Language Design and Implementation*, 2021. doi:10.1145/3453483.3454036.

23 Malte Schwerhoff and Alexander J. Summers. Lightweight support for magic wands in an automatic verifier. In *European Conference on Object-Oriented Programming*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.614.

24 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, 2009. doi:10.1007/978-3-642-03013-0_8.