Matching Logic Rewriting: Unifying Operational and Axiomatic Semantics in a Practical and Generic Framework

Grigore Roşu Andrei Ştefănescu
Department of Computer Science
University of Illinois at Urbana-Champaign
{grosu, stefane1}@illinois.edu

Abstract

Matching logic allows to specify structural properties about program configurations by means of special formulae, called *patterns*, and to reason about them by means of *pattern matching*. This paper proposes rewriting over matching logic formulae, which generalizes both term rewriting and Hoare triples, as a unified framework for operational semantics and for program verification. A programming language is formally defined as a set of rewrite rules. A *language-independent* nine-rule proof system then can be used either to derive any operational program behavior, or to verify programs against arbitrary properties specified also as rewrite rules, thus reducing the gap between operational semantics and axiomatic semantics to zero. The proof system is proved both sound and complete for operational semantics and partially correct for program verification. All these proofs are language-independent. A matching logic verifier for a fragment of C, called MatchC, has been implemented and evaluated with encouraging results on a series of non-trivial programs, attesting to the practicality of the approach.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

General Terms Languages, Semantics, Verification, Theory

Keywords Matching logic, operational semantics, axiomatic semantics, term rewriting

1. Introduction

An operational semantics defines an execution model of a language typically in terms of a transition relation $cfg \Rightarrow cfg'$ between program configurations, and can serve as a formal basis for language understanding, design, implementation, and so on. On the other hand, an axiomatic semantics defines a proof system typically in terms of Hoare triples $\{\psi\}\,\texttt{code}\,\{\psi'\}$, and can serve as a basis for program verification. To increase confidence in program verifiers, the programming language community has developed many techniques and tools for bridging the gap between operational and axiomatic semantics. Despite these, language designers still perceive the two kinds of semantics as two distinct endeavors, and proving their formal relationship as a burden. With very few notable exceptions (e.g., [1]), real languages are rarely given both semantics.

Technical report 2011 November 2011, Urbana, USA

One of the reasons for the disconnection between the two semantics is that operational semantics strongly rely on a notion of program configuration, saying how and where each piece of semantic information is structured and located, while axiomatic semantics deliberately attempt to avoid the program configurations, which are considered low-level, and instead capture their properties, including their structural properties, by means of logical formulae. This strong reliance on vs. deliberate abstraction of configurations put a 30-year-old wall between the two important semantic approaches.

Matching logic [47] is a logic designed to state and reason about structural properties over arbitrary program configurations. Syntactically, it introduces a new formula construct, called a *pattern*, which is a configuration term possibly containing variables. Semantically, its models are concrete/ground configurations, where a configuration satisfies a pattern iff it *matches* it. Considering a particular configuration structure with a top-level cell $\langle ... \rangle_{\mathsf{cfg}}$ holding, in any order, other cells with semantic data such as the code $\langle ... \rangle_{\mathsf{k}}$, an environment $\langle ... \rangle_{\mathsf{env}}$, a heap $\langle ... \rangle_{\mathsf{heap}}$, an input buffer $\langle ... \rangle_{\mathsf{in}}$, an output buffer $\langle ... \rangle_{\mathsf{out}}$, etc., configurations then have the structure:

$$\langle \langle ... \rangle_{\mathsf{k}}\ \langle ... \rangle_{\mathsf{env}}\ \langle ... \rangle_{\mathsf{heap}}\ \langle ... \rangle_{\mathsf{in}}\ \langle ... \rangle_{\mathsf{out}}\ \ ... \rangle_{\mathsf{cfg}}$$

The contents of the cells can be various semantic data, such as trees, lists, sets, maps, etc. Here are two particular configurations (in the interest of space, we use "..." for the irrelevant parts of them):

$$\langle \langle \texttt{x=*y; y=x;}\ ... \rangle_{\mathsf{k}}\ \langle \texttt{x} \mapsto 7,\ \texttt{y} \mapsto 3,\ ... \rangle_{\mathsf{env}}\ \langle 3 \mapsto 5 \rangle_{\mathsf{heap}}\ \ ... \rangle_{\mathsf{cfg}}$$
$$\langle \langle \texttt{x} \mapsto 3 \rangle_{\mathsf{env}}\ \langle 3 \mapsto 5,\ 2 \mapsto 7 \rangle_{\mathsf{heap}}\ \langle 1, 2, 3, ... \rangle_{\mathsf{in}}\ \langle ..., 7, 8, 9 \rangle_{\mathsf{out}}\ \ ... \rangle_{\mathsf{cfg}}$$

Different languages may have different configuration structures. For example, languages whose semantics are intended to be purely syntactic and based on substitution, e.g., λ-calculi, may contain only one cell, holding the program itself. Other languages may contain dozens of cells in their configurations; for example, the C semantics in [17] has more than 60 nested cells. However, no matter how complex a language is, its configurations can be defined as ground terms over an algebraic signature, using conventional algebraic techniques. Matching logic takes an arbitrary algebraic definition of configurations as parameter and allows configuration terms with variables as particular formulae. For example, the formula

$$\exists c{:}\mathit{Cells},\ e{:}\mathit{Env},\ p{:}\mathit{Nat},\ \ i{:}\mathit{Int},\ \sigma{:}\mathit{Heap}$$
$$\langle \langle \texttt{x} \mapsto p,\ e \rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma \rangle_{\mathsf{heap}}\ c \rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i$$

is satisfied by all configurations where program variable `x` points to a location p holding a different positive integer. The variables e, σ, and c are *structural frames*. If we want to additionally state that p is the only location allocated, then we can just remove σ:

$$\exists c{:}\mathit{Cells},\ e{:}\mathit{Env},\ p{:}\mathit{Nat},\ \ i{:}\mathit{Int}$$
$$\langle \langle \texttt{x} \mapsto p,\ e \rangle_{\mathsf{env}}\ \langle p \mapsto i \rangle_{\mathsf{heap}}\ c \rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i$$

Matching logic allows to reason about configurations, e.g., to show:

$$\begin{array}{ll}\models & \forall c{:}\mathit{Cells},\ e{:}\mathit{Env},\ p{:}\mathit{Nat} \\ & \quad \langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto 9\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ p > 10 \\ & \quad \rightarrow \exists i{:}\mathit{Int},\ \sigma{:}\mathit{Heap} \\ & \qquad \langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i\end{array}$$

In this paper we propose the novel concept of a

rewrite rule over matching logic formulae

and hereby *matching logic rewriting*. The semantics of a matching logic rewrite rule

$$\varphi \Rightarrow \varphi'$$

is that any configuration satisfying φ transits (in zero or more steps, depending on the context) into a configuration satisfying φ'. Such rewrite rules are quite expressive, subsuming the main elements of both operational and axiomatic semantics, namely both the usual rewrite (or reduction) rule and the Hoare triple, as explained next.

There is overwhelming evidence that languages and calculi can be given operational semantics based on rewrite (or reduction) rules of the form "$l \Rightarrow r$ if b", where l and r are configuration terms with variables constrained by boolean condition b. Tools, techniques and methodologies supporting such operational semantics, like Redex [19] and $\mathbb{K}$ [44] among others, as well as large languages defined using these (e.g., the C semantics [17] exceeds 1,000 such rules), stand as proof that this is not only possible, but also practical. Such rules can be expressed as matching logic rewrite rules $l \wedge b \Rightarrow r$, allowing to regard operational semantics following these approaches as matching logic rewrite systems.

On the other hand, a Hoare triple of the form $\{\psi\}\ \mathtt{code}\ \{\psi'\}$ can be regarded as a matching logic rewrite rule $\langle \mathtt{code}\rangle_{\mathsf{k}} \wedge \psi \Rightarrow \langle\rangle_{\mathsf{k}} \wedge \psi'$ between formulae over minimal configurations holding only the code. Here, $\langle\rangle_{\mathsf{k}}$ is the configuration holding the empty code, so the rule asserts that ψ' holds when/if code terminates.

Therefore, both operational semantics rules and axiomatic semantics Hoare triples are instances of matching logic rules. Could we then use matching logic rewriting as a unifying framework for operational and axiomatic semantics? Or, more specifically, could it be possible to start with an operational semantics of a programming language defined as a matching logic rewrite system, and then derive program properties also expressed as matching logic rules without a need to give the language another semantics?

The main contribution of our paper is the nine-rule language-independent proof system for matching logic rewriting in Figure 1. Reflexivity and Transitivity are inspired by rewriting logic [35]. Case analysis, Logic framing, Consequence and Abstraction are inspired by Hoare logic [25]. Axiom and Substitution by both. The Circularity proof rule is new. It deductively and language-independently captures the various circular behaviors that appear in languages, due to loops, recursion, jumps, etc. $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ means that the matching logic rule $\varphi \Rightarrow \varphi'$ is derivable from a set of matching logic rules $\mathcal{A}$ using all nine proof rules, while $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'$ means that $\varphi \Rightarrow \varphi'$ is derivable from $\mathcal{A}$ using all proof rules but Reflexivity, indicating that at least one proper semantic step is taking place.

A programming language operational semantics is given as a set of rewrite rules, which is the original $\mathcal{A}$. Indeed, any matching logic rewrite system $\mathcal{A}$ yields a transition system over ground configurations, with transitions between two ground configurations γ and γ' if and only if there is some rule $\varphi \Rightarrow \varphi'$ in $\mathcal{A}$ such that γ matches φ and γ' matches φ', respectively. This transition system therefore captures all the concrete *operational behaviors* of the target programming languages. Our proof system in Figure 1 can be then used either to generate such concrete, operational program behaviors (the first eight proof rules), or to prove program properties specified as matching logic rules. During the proof derivation, one may add new rules to $\mathcal{A}$ by means of Circularity, which are thus allowed to be used in their own derivation! The correctness of this proof circularity is given by the fact that progress is required to be made (indicated by $\Rightarrow^{+}$ in $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'$) before a circular reasoning step is allowed.

Rules of operational nature

Reflexivity: $$\frac{}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

Axiom: $$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Substitution: $$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad \theta : \mathit{Var} \to \mathcal{T}_\Sigma(\mathit{Var})}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

Transitivity: $$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}$$

Rules of deductive nature

Case analysis: $$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Logic framing: $$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad \psi \text{ is a FOL}_{=} \text{ formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence: $$\frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Abstraction: $$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash \exists X\ \varphi \Rightarrow \varphi'}$$

Rule for circular behavior

Circularity: $$\frac{\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'' \quad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Figure 1. Matching logic rewriting proof system

We prove the first eight rules of the proof system *sound* and *complete* for operational deduction, in the sense that any ground rewrite property derived with these eight rules corresponds indeed to an operational behavior, and that any operational behavior can be derived also using these eight rules (Theorem 1). Then the entire nine-rule proof system is proved *partially correct* (Theorem 2), in the sense that any derived property of an operationally *terminating* ground configuration corresponds to an operational behavior.

To our knowledge, this is the first proof system of its kind. A language designer now only has to define *one* rewriting semantics of the target programming language, which is well-understood, tool-supported and comparatively easier than defining an axiomatic semantics. Moreover, the rewriting semantics is executable and thus testable using existing rewrite engines (some quite efficient) or functional languages incorporating term rewriting (e.g., Haskell). For example, one can test it by executing program benchmarks that compiler testers use. Then, one can take this semantics and use it *as is* for program verification. Not only that one can now completely skip the tedious step of having to prove the equivalence between an operational and an axiomatic semantics of the same language, but one can also change the language at will (or fix it when semantic bugs are found), without having to worry about doing that in two different places and maintaining the equivalence proofs.

The main concern to a verification framework based on operational semantics is that it may not be practical, due to the amount of required user involvement or to the amount of low-level details

```
struct listNode { int val; struct listNode *next; };

int main()
{
  struct listNode *x;
  x = (struct listNode*) malloc(sizeof(struct listNode));
  printf("%p\n", x->next);
}
```

Figure 2. C program exhibiting undefined behaviour.

that needs to be provided in specifications. To test the practical effectiveness of matching logic rewriting, we implemented a new version of the MatchC program verifier for a fragment of C. Previously, MatchC followed a Hoare-style approach using matching logic formulae for invariants and for pre- and post-conditions [45]. The new MatchC uses matching logic rewrite rules for program specifications and its implementation is directly based on the proof system in Figure 1. It uses the rewrite semantics of the fragment of C *completely unchanged* for program verification. The Matching Logic web page, `http://fsl.cs.uiuc.edu/ml`, contains an online interface to run MatchC. The online interface includes all the examples mentioned in this paper, among others, allowing users to modify and verify them or to type in new programs.

Specific contributions:

- A more foundational and general formulation of matching logic;
- Matching logic rewriting, with the sound, complete and partially correct language-independent proof system Figure 1;
- Implementation and evaluation of the new MatchC.

Paper structure: Section 2 discusses several programs verified using MatchC. Section 3 gives a more thorough and general formalization of matching logic than in previous work. Section 4 presents our main theoretical results, and Section 5 discusses our implementation. Section 6 discusses related work. Section 7 concludes and discusses future work. The proofs are all exiled in Appendix A.

2. Motivating Examples using MatchC

Here we discuss a few C examples that illustrate the expressiveness and practicality of matching logic rewriting. Figure 2 shows an undefined program. Figure 3 a function that reverses a singly linked list. Figure 4 a function that reads a sequence of integers from the standard input into a singly-linked list. Figure 5 a program that respects a stack inspection property, where some functions can only be called directly or indirectly by certain other functions, and only under certain conditions. Figure 6 shows a function that flattens a tree into a list, traversing the tree in infix order and in the process printing the list to the standard output in reverse order. MatchC automatically verifies all these programs in ~1s in total (Section 5).

The unannotated/unspecified program in Figure 2 is undefined according to the C standard, because it attempts to print the value of the uninitialized list member `next`. Our rewriting semantics of the C fragment correctly captures undefinedness, in that undefined programs get stuck during their execution using the semantics. MatchC verifies programs by executing them using the semantic rewrite rules. If a fragment of code is given a specification, then that specification is verified and subsequently used as a replacement for the corresponding fragment. This is possible in matching logic because both the language semantics and the specifications are uniformly given as rewrite rules. Since this program is unannotated, its matching logic verification reduces to executing it in the semantics, so it gets stuck when reading `x->next`. C compilers happily compile this program and the generated code even does what one (wrongly) expects it to do, namely prints the residual value of `x->next`.

```
struct listNode { int val; struct listNode *next; };

struct listNode* reverseList(struct listNode *x)
rule ⟨$ ⇒ return ?p; ···⟩k ⟨··· list(x)(A) ⇒ list(?p)(rev(A)) ···⟩heap
{
  struct listNode *p, *y;
  p = NULL;
inv ⟨··· list(p)(?B), list(x)(?C) ···⟩heap ∧ A = rev(?B)@?C
  while(x != NULL) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

Figure 3. C function reversing a singly-linked list.

Some MatchC notations. Each rule or invariant user annotation (grayed area in the figures in this section) corresponds to a matching logic rule, also called a specification, that needs to be derived using the proof system in Figure 1 from the rewriting semantics of the language. To facilitate understanding the next specifications, we briefly discuss some MatchC notations that help avoid verbosity:

- While all specifications are rewrite rules $\varphi \Rightarrow \varphi'$ between matching logic formulae, often φ and φ' share configuration context; we only mention the context once and distribute the "$\Rightarrow$" arrow through the context where the changes take place.
- To avoid writing existential quantifiers, logical variables starting with "?" are assumed existentially quantified.
- To avoid writing environment cells containing only bindings of the form $\mathtt{x} \mapsto ?\mathsf{x}$ in almost all specifications, we automatically assume them when not explicitly mentioned and allow users to write the identifier `x` (which is a syntactic constant) instead of the logical variable ?x.
- MatchC desugars invariants `inv` φ `loop` into matching logic proof obligation rules $\varphi[\mathtt{loop...}] \Rightarrow \varphi[...] \wedge \neg \mathit{cond}(\mathtt{loop})$, where $\varphi[\mathtt{code}]$ is the pattern obtained from φ by replacing the contents of the $\langle ... \rangle_k$ cell with `code`.

Function `reverseList` in Figure 3 reverses a singly-linked list. The matching logic rule specifying its behavior says that it returns a pointer ?p (here and in the rest of the paper, $ stands for the body of the function). The rule also specifies that, when the function is called, the heap contains a list starting at `x` with contents the sequence A. By the time the function returns, the initial list is replaced by a list starting at ?p with contents the reversed sequence, rev(A). The ··· in the heap cell stands for the rest of the heap content (the *heap frame*) which is not touched by the function and thus stays unchanged. Similarly, all the parts of the configuration that are not explicitly mentioned (the *configuration frame*) do not change. The loop invariant asserts that the heap contains two lists, one starting at `p` and containing the part of the sequence that is already reversed, ?B, and one starting at `x` and containing the part of the sequence that is yet to be reversed, ?C. The initial sequence A equals rev(?B) followed by ?C. Again, the rest of the heap and of the configuration stay unchanged. In matching logic, list, rev, etc., are ordinary *operation* symbols in the signature and constrained through axioms (see Section 3). Like in OCaml, @ concatenates sequences. Variables without ?, like A, are free. Hence, A refers

```
struct listNode { int val; struct listNode *next; };

struct listNode *readList(int n)
```

rule ⟨$ ⇒ return ?x; ···⟩k ⟨A ⇒ · ···⟩in ⟨··· · ⇒ list(?x)(A) ···⟩heap
if n = len(A)

```
{
  int i;
  struct listNode *x, *p;
  if (n == 0) return NULL;
  x = (struct listNode*) malloc(sizeof(struct listNode));
  scanf("%d", &(x->val));
  x->next = NULL;
  i = 1; p = x;
```

inv ⟨?C ···⟩in ⟨··· lseg(x, p)(?B), p ↦ [?v, NULL] ···⟩heap
∧ i ≤ n ∧ len(?C) = n − i ∧ A = ?B@[?v]@?C

```
  while (i < n) {
    p->next = (struct listNode*)
              malloc(sizeof(struct listNode));
    p = p->next;
    scanf("%d", &(p->val));
    p->next = NULL;
    i += 1;
  }
  return x;
}
```

Figure 4. C function reading a sequence of integers from the standard input into a singly-linked list.

```
void trusted(int n);
void untrusted(int n);
void any(int n);

void trusted(int n)
```

rule ⟨$ ⇒ return; ···⟩k ⟨S⟩stack
if n ≥ 10 ∨ in(hd(ids(S)), [main, trusted])

```
{
  untrusted(n); any(n);
  if (n) trusted(n - 1);
}

void untrusted(int n)
```

rule ⟨$ ⇒ return; ···⟩k⟨S⟩stack
if in(trusted, ids(S))

```
{
  if (n) any(n - 1);
}

void any(int n)
{
  // possible security policy violation
  // (when any is called) if n <= 10
  if(n > 10) trusted(n - 1);
}

int main() { trusted(5); any(5); }
```

Figure 5. C program respecting a stack inspection security policy.

to the same sequence in the function rule and in the loop invariant, while ?B can refer to different sequences in different loop instances.

Function `readList` in Figure 4 reads `n` integers from standard input and stores them in a singly-linked list. The matching logic specification says that the function: (1) returns a pointer ?x; (2) reads from the standard input a sequence of integers A of length `n` (matches A and replaces it by the empty sequence ·); (3) allocates a list starting at ?x with contents the read sequence A (replaces the empty heap ·). As before, the rest of the input buffer, the heap, and the configuration stay unchanged. The loop invariant states that the sequence ?C is yet to be read, `x` points to a list segment ending at `p` with contents ?B, `p` points to a `nodeList` structure with the value field ?v and the `next` field `NULL`, the loop index `i` is not greater than `n`, the size of ?C is `n − i`, and the initial sequence A equals the concatenation of ?B, ?v, and ?C. The list segment lseg(`x`, `p`) includes `x` but excludes `p`. The notation `p` ↦ [?v, `NULL`] stands for the *term* (and *not* formula) "`p` ↦ ?v, `p` + 1 ↦ `NULL`".

Figure 5 shows a C program that respects the following security policy: `trusted` must always be called directly with the argument `n`'s value less than 10 only from `main`, or from `trusted` (suppose that `n` represents some priority or clearance level), while `untrusted` must always be called directly or indirectly from `trusted` (suppose that `trusted` is the only function whose code is completely trusted, so in particular it is even allowed to call untrusted functions). The matching logic specification rule of `trusted` matches the call stack, and requires that either the value of `n` is at least 10, or that the function id of the head of the call stack is one of `main` or `trusted`. The rest of the configuration stays unchanged. The rule for `untrusted` matches the same parts of the configuration as the rule for `trusted`, but requires instead that somewhere in the call stack there exists a frame for `trusted`. Function `any` does not have a rule, so its body is executed at each call. Note that if the call to `trusted` in `any` were not guarded by the if statement, the line `any(5);` in `main` would violate the security policy. Just constructing the call graph and performing value analysis is not enough to verify these stack properties.

Function `treeToList` in Figure 6 flattens a binary tree into a list, by traversing the tree in infix order, and in the process prints the list to the standard output in reverse order. Each node of the initial tree (structure `treeNode`) has three fields: the value, and two pointers, for the left and the right subtrees. Each node of the final list (structure `listNode`) has two fields: the value and a pointer to the next node of the list. The program makes use of an auxiliary structure (`stackNode`) to represent a stack of trees. For demonstration purposes (to highlight the invariant capability of matching logic), we prefer an iterative version of this program. We need a stack to keep track of our position in the tree. Initially, that stack contains the tree passed as argument (as a pointer). The loop repeatedly pops a tree from the stack, and it either pushes back the left tree, the root, and the right tree onto the stack, or if the right tree is empty it pushes back the left subtree, appends the value in the root node at the beginning of the list of tree elements, and prints the respective value to the standard output. As the loop processes the tree, it frees the tree nodes and it allocates the corresponding list nodes. Because the values are printed when they are popped from the stack, they appear in the output in reverse infix order.

The `treeToList` rule specifies that it returns a pointer ?l. It matches in the heap a tree rooted at `t` holding the contents T, and replaces it with a list starting at ?l with the contents tree2list(T) (the infix traversal sequence of T). Finally, it specifies that the function outputs the traversal sequence in reverse order. The rest of the heap, output buffer and the configuration stay unchanged. The invariant says that the heap contains a stack of trees (represented as a list of trees) with contents ?TS and a list with contents ?A, the loop has printed so far the sequence rev(?A), and that the infix traversal sequence of T, tree2list(T), is equal to the concatenation in reverse order of the infix traversal sequences of the trees in the stack concatenated with the contents of the list. Nothing else changes.

```
struct treeNode {
  int val;
  struct treeNode *left, *right;
};
struct listNode {
  int val;
  struct listNode *next;
};
struct stackNode {
  struct treeNode *val;
  struct stackNode *next;
};

struct listNode *treeToList(struct treeNode *t)
```

rule ⟨$ ⇒ **return** ?l; ···⟩$_k$ ⟨··· tree(x)(T) ⇒ list(?l)(tree2list(T)) ···⟩$_{heap}$
⟨··· · ⇒ rev(tree2list(T))⟩$_{out}$

```
{
  struct listNode *l; struct stackNode *s;
  if (t == NULL) return NULL;

  l = NULL;
  s = (struct stackNode *)
      malloc(sizeof(struct stackNode));
  s->val = t;
  s->next = NULL;
```

inv ⟨··· tree(s)(?TS), list(l)(?A) ···⟩$_{heap}$ ⟨··· rev(?A)⟩$_{out}$
∧ tree2list(T) = treeList2list(rev(?TS))@?A

```
  while (s != NULL) {
    struct treeNode *tn; struct listNode *ln;
    struct stackNode *sn;
    sn = s;
    s = s->next;
    tn = sn->val;
    free(sn) ;
    if (tn->left != NULL) {
      sn = (struct stackNode *)
           malloc(sizeof(struct stackNode));
      sn->val = tn->left;
      sn->next = s;
      s = sn;
    }
    if (tn->right != NULL) {
      sn = (struct stackNode *)
           malloc(sizeof(struct stackNode));
      sn->val = tn;
      sn->next = s;
      s = sn;
      sn = (struct stackNode *)
           malloc(sizeof(struct stackNode));
      sn->val = tn->right;
      sn->next = s;
      s = sn;
      tn->left = tn->right = NULL;
    }
    else {
      ln = (struct listNode *)
           malloc(sizeof(struct listNode));
      ln->val = tn->val;
      ln->next = l;
      l = ln;
      printf("%d␣", ln->val);
      free(tn);
    }
  }
  return l;
}
```

Figure 6. Iterative C program flattening a tree into a list and printing its values in the process.

3. Matching Logic: A Logic of Configurations

Matching logic was introduced in [47], although the focus there was more on presenting an axiomatic semantic framework based on matching logic, and less on presenting matching logic itself as a logic for reasoning about configurations. Since here we need matching logic for both axiomatic and operational semantics, in a style quite different from that in [47] requiring a more fundamental understanding of matching logic, we use this as an opportunity to give a thorough and self-contained presentation of matching logic, which also slightly generalizes the one in [47]. Specifically, we allow more matching logic specifications than the patterns of [47].

3.1 Algebraic and Logical Background

We remind the reader of basic concepts of algebraic specification and first-order logic. The role of this section is also to establish our notation used later in the paper. We refer the reader to [12, 37] and references there for a thorough and practical introduction to these concepts in the context of systems like CASL and Maude.

An *algebraic signature* $(\mathbb{S}, \Sigma)$ is a finite set of *sorts* $\mathbb{S}$ and a finite set of *operation symbols* Σ over sorts in $\mathbb{S}$. We may write Σ instead of $(\mathbb{S}, \Sigma)$. A Σ-*algebra* is an $\mathbb{S}$-indexed set together with functions corresponding to operation symbols. $\mathcal{T}_\Sigma$ denotes the *initial* Σ-*algebra of ground terms* (i.e., terms without variables). $\mathcal{T}_\Sigma(X)$ denotes the *free* Σ-*algebra of terms with variables in* X, where X is an $\mathbb{S}$-indexed set of variables. $\mathcal{T}_{\Sigma,s}(X)$ denotes the set of Σ-terms of sort $s \in \mathbb{S}$. A Σ-*equation*[1] is a triple $\forall X(t = t')$, where $t, t' \in \mathcal{T}_{\Sigma,s}(X)$ and $s \in \mathbb{S}$. Σ-algebra M *satisfies* Σ-equation $\forall X(t = t')$, written $M \models \forall X(t = t')$, iff $\theta(t) = \theta(t')$ for any $\theta : X \to M$ (we write also θ for its homomorphic extension). If $\mathcal{E}$ is a set of equations, $M \models \mathcal{E}$ means $M \models e$ for each $e \in \mathcal{E}$. $\mathcal{E} \models e$ means that $M \models \mathcal{E}$ implies $M \models e$ for any Σ-algebra M. An *algebraic specification* $(\mathbb{S}, \Sigma, \mathcal{E})$ is an algebraic signature $(\mathbb{S}, \Sigma)$ together with a set of Σ-equations $\mathcal{E}$.

For example, $\mathbb{S}$ may include sorts *Exp* for expressions and *Stmt* for statements, and Σ may include operation symbols like

$$\begin{array}{ll} \{\} : \to \mathit{Stmt} & \text{(empty block)} \\ \texttt{if(\_)\_} : \mathit{Exp} \times \mathit{Stmt} \to \mathit{Stmt} & \\ \texttt{if(\_)\_else\_} : \mathit{Exp} \times \mathit{Stmt} \times \mathit{Stmt} \to \mathit{Stmt} & \end{array}$$

We used the mixfix notation above, with underscores as argument placeholders, which is equivalent to the context-free or BNF notation. Since the latter is more common for defining programming language syntax, we prefer it from here on; so we write "*Stmt* ::= {} | `if` (*Exp*) *Stmt* | `if` (*Exp*) *Stmt* `else` *Stmt*". An example of a Σ-equation is the desugaring of `if(_)_` into `if(_)_else_`:

$$\forall E : \mathit{Exp},\ S : \mathit{Stmt}\ (\texttt{if}\,(E)\,S\ =\ \texttt{if}\,(E)\,S\ \texttt{else}\,\{\})$$

Thanks to the completeness of equational and of first-oder logics, $\mathcal{E} \models e$ is equivalent to saying that e can be derived from $\mathcal{E}$ using equational or first-order deduction. We do not recall these deduction systems here. The notions of initial and free algebras extend to algebraic specifications, through factorization by the equivalence classes of terms generated by the equations. Given algebraic specification $(\mathbb{S}, \Sigma, \mathcal{E})$, we let $\mathcal{T}_{\Sigma/\mathcal{E}}$ be its *initial algebra* of ground terms and let $\mathcal{T}_{\Sigma/\mathcal{E}}(X)$ be its *free algebra* of terms with variables in X. $\mathcal{T}_{\Sigma,\mathcal{E}}$ captures all the *ground* equational properties of $\mathcal{E}$, in that for any ground Σ-equation e, $\mathcal{T}_{\Sigma,\mathcal{E}} \models e$ iff $\mathcal{E} \models e$. This implies that ground equational satisfaction in the initial model is semi-decidable. Note that this is *not* true for equations with variables [33]. One of the most fundamental results in algebraic specification is due to Bergstra and Tucker [7] and essentially says that any semi-decidable domain is isomorphic to an initial algebra of a finite algebraic specification.

[1] In the interest of space, we only show unconditional equations here.

There is a plethora of algebraic specification variants (order-sorted, membership, partial, etc.) as well as a vast literature showing how to define almost every known mathematical or computing structure as a Σ-algebra over an appropriate algebraic specification, including boolean algebras, natural/integer/rational numbers, monoids, groups, rings, lists, sets, bags (or multisets), mappings, trees, queues, stacks, and so on, as well as combinations of them. Systems like CASL [37] and Maude [12] use algebraic specifications as their underlying semantic infrastructure; we refer the reader to their manuals for state of the art and examples.

From here on we take the freedom to use common structures like lists, sets, bags, and maps over any sorts, including other lists, sets, etc., by simply mentioning their sorts as parameters. Also, if S and S' are sorts, we write $S \times S'$ for their product sort. For example, $Map_{Bag_{Nat},Int \times Int}$ is the sort corresponding to maps taking bags of naturals to pairs of integers. For notational simplicity, we (ambiguously) use a central dot "$\cdot$", read "nothing", for the units of all lists, sets, bags, maps, etc., a comma "," or a whitespace "$\sqcup$" for their concatenation, and an infix "$\mapsto$" for building maps. We also use parentheses for grouping. For example, "$3 \mapsto (-1, 1),\ \cdot \mapsto (7, -3),\ 3\ 5\ 3 \mapsto (-7, -2)$" is a term of sort $Map_{Bag_{Nat},Int \times Int}$, which is equal to "$3\ 5\ 3 \mapsto (-7, -2),\ 3 \mapsto (-1, 1),\ \cdot \mapsto (7, -3)$".

We next briefly recall *first-order logic with equality (FOL$_=$)*, which extends algebraic specifications. A *first-order signature* $(\mathbb{S}, \Sigma, \Pi)$ extends an algebraic signature $(\mathbb{S}, \Sigma)$ with a finite set of predicate symbols Π. FOL$_=$ formulae have the syntax

$$\psi ::= t = t' \mid pred(t_1, ..., t_n) \mid \forall X\ \psi \mid \psi_1 \wedge \psi_2 \mid \neg\psi$$

plus the usual derived constructs $\psi_1 \vee \psi_2$, $\psi_1 \rightarrow \psi_2$, $\exists X\ \psi$, etc., where $t, t', t_1, ..., t_n$ range over Σ-terms of appropriate sorts, $pred \in \Pi$ over atomic predicates, and X over finite $\mathbb{S}$-indexed sets of variables. Σ-terms can have variables; all variables are chosen from a fixed sort-wise infinite $\mathbb{S}$-indexed set of variables, *Var*. A *FOL$_=$ specification* $(\mathbb{S}, \Sigma, \Pi, \mathcal{F})$ is a FOL$_=$ signature $(\mathbb{S}, \Sigma, \Pi)$ plus a set of *closed* (i.e., no free variables) formulae $\mathcal{F}$. A *FOL$_=$ model M* is a Σ algebra together with relations for the predicate symbols in Π. Given any closed formula ψ and any model M, we write $M \models \psi$ iff M satisfies ψ. If ψ has free variables and $\rho : Var \rightarrow M$, also called an *M-valuation*, we let $\rho \models \psi$ denote the fact that ρ satisfies ψ.

3.2 Configurations

Matching logic is parametric in configurations, more precisely in a model for configurations. In this section we discuss such a model, noting that different programming languages or calculi typically have different configurations. The same machinery works for all.

Figure 7 shows the configuration syntax of the C fragment discussed in this paper; our C fragment here is almost identical to the KernelC language formally defined in [46]. The maps, bags and lists over the various sorts can be defined using conventional algebraic specification techniques, as discussed in Section 3.1. Sorts *Nat* and *Int* come with various domain operations on them, which can be used for reasoning. We only consider integer, structure and pointer types in our fragment. The sort K is a generic sort for "code" and comprises the entire syntax of the programming language; thus, terms of sort K correspond to fragments of program. Environments are terms of sort *Env* and are maps from identifiers to integers. Type environments in *TEnv* map identifiers to types.

A program configuration for our particular language is a term of sort *Cfg* of the form $\langle ... \rangle_{\mathsf{cfg}}$ containing a bag of semantic cells. In addition to $\langle ... \rangle_{\mathsf{k}}$, $\langle ... \rangle_{\mathsf{env}}$ and $\langle ... \rangle_{\mathsf{tenv}}$ holding a fragment of a program, an environment and a type environment, respectively, $\langle ... \rangle_{\mathsf{cfg}}$ also holds the following cells: $\langle ... \rangle_{\mathsf{struct}}$ holding the available data structures as a map from structure names to lists of typed fields; $\langle ... \rangle_{\mathsf{funs}}$ holding the available functions as a map from function names to their arguments and body; $\langle ... \rangle_{\mathsf{fname}}$ holding the name of the current function; $\langle ... \rangle_{\mathsf{stack}}$ holding the function stack as a list of frames, each frame containing a function name and its execution context (the remaining code, the environment and the type environment); $\langle ... \rangle_{\mathsf{heap}}$ holding the heap as a map from natural numbers (pointers) to integers (values); $\langle ... \rangle_{\mathsf{in}}$ holding the input buffer as a list of integers; and $\langle ... \rangle_{\mathsf{out}}$ holding the output buffer also as a list of integers.

Id	::=	C identifiers
Nat	::=	domain of natural numbers (including operations)
Int	::=	domain of integer numbers (including operations)
Type	::=	`int` | `struct` *Id* | *Type* `*`
K	::=	the entire remaining syntax of the C fragment
Env	::=	$Map_{Id,Int}$
TEnv	::=	$Map_{Id,Type}$
Cell	::=	$\langle Map_{Id,List_{Type \times Id}} \rangle_{\mathsf{struct}}$
	|	$\langle Map_{Id,List_{Type \times Id} \times K} \rangle_{\mathsf{funs}}$
	|	$\langle K \rangle_{\mathsf{k}}$
	|	$\langle Env \rangle_{\mathsf{env}}$
	|	$\langle TEnv \rangle_{\mathsf{tenv}}$
	|	$\langle Id \rangle_{\mathsf{fname}}$
	|	$\langle List_{Id \times K \times Env \times TEnv} \rangle_{\mathsf{stack}}$
	|	$\langle Map_{Nat,Int} \rangle_{\mathsf{heap}}$
	|	$\langle List_{Int} \rangle_{\mathsf{in}}$
	|	$\langle List_{Int} \rangle_{\mathsf{out}}$
Cfg	::=	$\langle Bag_{Cell} \rangle_{\mathsf{cfg}}$

Figure 7. Sample configuration

Our only reason for choosing the particular configuration structure in Figure 7 is because it turned out to be good enough to define a formal executable semantics for our language (this is further discussed in Section 4.1). While matching logic can be used for program verification without a formal semantics of the language, but only with an axiomatic Hoare-style semantics [47], as seen in Section 4 one of its major advantages is that it can also be used in combination with a formal semantics of the language, without a need to redefine the language semantics axiomatically. We therefore strongly advocate having a formal semantics of the language in order to do matching logic verification. That not only gives us an immediate definition of configurations, like in Figure 7, but it can also be used as is for program verification, as shown in Section 4.3.

Let Σ be the algebraic signature associated to some desired configuration syntax. Then a Σ-algebra gives us a configuration *model*, namely a universe of concrete language configurations. Moreover, Bergstra and Tucker's result [7] tells us that no matter what model of configurations one prefers for a language in order to give that language a formal semantics, there is some finite algebraic specification $(\Sigma, \mathcal{E})$ whose initial model is isomorphic to that model. In practice, however, one may prefer different means to define a model for configurations, for example using first-order or higher-order logics, or even directly, using informal mathematics.

From here on we simply assume that $\mathcal{T}$ is a given configuration model, that is, a Σ-algebra, where Σ is the algebraic signature of configurations. We do not impose any particular means to define $\mathcal{T}$. In practice, one will likely choose a reasonable model as well as a background logical theory that allows one to reason about it. For example, with the configurations in Figure 7, one may want to be able to infer that any two locations in the heap are distinct:

$$\mathcal{T} \models \forall cfg : Cfg, c : Bag_{Cell}, n : Nat, m : Nat, i : Int, j : Int, \sigma : Map_{Nat,Int} \\ (cfg = \langle c\ \langle n \mapsto i,\ m \mapsto j,\ \sigma \rangle_{\mathsf{heap}} \rangle_{\mathsf{cfg}}) \rightarrow n \neq m$$

or even more compactly

$$\mathcal{T} \models \forall n : Nat, m : Nat, i : Int, j : Int, \sigma : Map_{Nat,Int} \\ (\sigma = n \mapsto i,\ m \mapsto j) \rightarrow n \neq m$$

Similarly, in order to state properties like the ones in Section 2, one has to ensure that Σ contains operator symbols corresponding to lists of integer numbers and append and reverse on them, for membership testing of integers to such lists, for binary trees of integer numbers and for flattening such trees into lists, as well as for lists of trees and their flattening into lists, etc. Moreover, in order to verify the programs in Section 2, MatchC needed all the equational properties below, so the configuration model $\mathcal{T}$ underlying MatchC satisfies all of them. To avoid notational clutter, we assume that all the variables in the equations below are universally quantified:

$$\begin{aligned}
\mathsf{rev}(\mathsf{nil}) &= \mathsf{nil}\\
\mathsf{rev}([\mathsf{a}]) &= [\mathsf{a}]\\
\mathsf{rev}(\mathsf{A}_1 @ \mathsf{A}_2) &= \mathsf{rev}(\mathsf{A}_2) @ \mathsf{rev}(\mathsf{A}_1)\\
\mathsf{in}(\mathsf{a}, \mathsf{nil}) &= \mathsf{false}\\
\mathsf{in}(\mathsf{a}, [\mathsf{b}]) &= (\mathsf{a} == \mathsf{b})\\
\mathsf{in}(\mathsf{a}, \mathsf{A}_1 @ \mathsf{A}_2) &= \mathsf{in}(\mathsf{a}, \mathsf{A}_2) \vee \mathsf{in}(\mathsf{a}, \mathsf{A}_1)\\
\mathsf{tree2list}(\mathsf{empty}) &= \mathsf{nil}\\
\mathsf{tree2list}(\mathsf{tree}(\mathsf{a}, \tau_l, \tau_r)) &= \mathsf{tree2list}(\tau_l) @ [\mathsf{a}] @ \mathsf{tree2list}(\tau_r)\\
\mathsf{treeList2list}(\mathsf{nil}) &= \mathsf{nil}\\
\mathsf{treeList2list}([\tau]) &= \mathsf{tree2list}(\tau)\\
\mathsf{treeList2list}(\mathsf{A}_1 @ \mathsf{A}_2) &= \mathsf{treeList2list}(\mathsf{A}_1) @ \mathsf{treeList2list}(\mathsf{A}_2)
\end{aligned}$$

3.3 Matching Logic

Traditionally, program logics are deliberately not concerned with low-level details pertaining to program configurations, those details being almost entirely deferred to operational semantics. This is a lost opportunity, since configurations contain very precious information about the *structure* of the various data in a program's state, such as the heap, the stack, the input, the output, etc. Without direct access to this information, program logics end up having to either encode it by means of sometimes hard to define predicates, or extend themselves in non-conventional ways, or sometimes both. In contrast, matching logic takes program configurations at its core. We follow a first-order approach here since we found it sufficient until now in our experiments, but it can be adapted to any logical formalism that provides support for signatures and terms.

As discussed in Section 3.2, we assume given a syntax and a model for configurations. More precisely, let Σ be an algebraic signature (configuration syntax) and let $\mathcal{T}$ be a Σ-algebra (configuration model). Σ includes, in particular, the entire abstract language syntax and the abstract syntax of all the data-structures that one may need in order to give the language an operational semantics over the given configurations. To simplify the presentation, let us assume Σ has a distinguished sort *Cfg* (for configurations).

We next present matching logic as an extension of $\mathrm{FOL}_{=}$ over Σ, but then we show that it easily translates back into $\mathrm{FOL}_{=}$. In other words, syntactically speaking, matching logic is a methodological fragment of $\mathrm{FOL}_{=}$. Semantically, its satisfaction is defined in terms of the chosen configuration model, $\mathcal{T}$, so in theory one can prove more properties in matching logic than in $\mathrm{FOL}_{=}$. In practice, it is expected that sufficiently many properties/axioms of $\mathcal{T}$ are available so that one can identify $\mathcal{T}$ with its properties. Nevertheless, since $\mathcal{T}$ is likely to be an initial model, or close to one, one should not expect complete $\mathrm{FOL}_{=}$ axiomatizations of $\mathcal{T}$ in general [33].

3.3.1 Syntax

Syntactically, matching logic extends the formulae of $\mathrm{FOL}_{=}$ over Σ with special formulae, called *patterns*, which are nothing but Σ-terms of sort *Cfg* which can contain variables. One can make it more general and allow patterns of any sort, but for simplicity here we limit ourselves to just *Cfg* patterns. Let *Var* be a sort-wise infinite set of variables. Unlike in Hoare logic, in matching logic there is no relationship between program variables and logical variables; with the particular configuration in Section 3.2, the former are actually constants of sort *Id*. Recall from Section 3.1 that $\mathcal{T}_\Sigma$ is the set of ground (i.e., no variables) Σ-terms and that $\mathcal{T}_\Sigma(\mathit{Var})$ is the Σ-algebra of terms with variables in *Var*, and from Section 3.2 that there is no enforced relationship between $\mathcal{T}$ and $\mathcal{T}_\Sigma$ (or $\mathcal{T}_\Sigma(\mathit{Var})$).

Definition 1. *Matching logic extends the syntax of $\mathrm{FOL}_{=}$ by adding Σ-terms with variables, called **basic patterns**, as formulae:*

$$\varphi \quad ::= \quad \textit{... conventional } \mathrm{FOL}_{=} \textit{ syntax} \quad | \quad \mathcal{T}_{\Sigma,\mathit{Cfg}}(\mathit{Var})$$

*Matching logic formulae of the form $\pi \wedge \psi$ with π a basic pattern and ψ a standard $\mathrm{FOL}_{=}$ formula (with no patterns) are called **constrained patterns**, ones of the form $\exists X \pi$ with $X \subset \mathit{Var}$ and π a constrained pattern are called **existential patterns**, and ones of the form $\pi_1 \vee ... \vee \pi_n$ with each π_i an existential pattern are called **disjunctive patterns**. We call all the above generically **patterns**.*

We let ψ, ψ_1, ψ', etc., range over $\mathrm{FOL}_{=}$ formulae (containing no patterns), π, π_1, π', etc., over patterns, and φ, φ_1, φ', etc., over arbitrary matching logic formulae (typically containing patterns).

Patterns are the most common matching logic formulae. We next give some examples of patterns. For concreteness, unless otherwise specified, from here on in the paper we work with the particular signature and model $\mathcal{T}$ of configurations in Section 3.2.

Given program variable $\mathtt{x}$ (i.e., a constant of sort *Id*), the pattern

$$\exists c{:}\mathit{Bag}_{\mathit{Cell}},\ e{:}\mathit{Env}\ \langle\langle \mathtt{x} \mapsto 5,\ e\rangle_{\mathsf{env}}\ c\rangle_{\mathsf{cfg}}$$

specifies (the exact semantics of matching logic will be given shortly) those program configurations in which $\mathtt{x}$ is bound to 5 in the environment. Similarly, the pattern

$$\exists c{:}\mathit{Bag}_{\mathit{Cell}},\ e{:}\mathit{Env},\ i{:}\mathit{Int}\ (\langle\langle \mathtt{x} \mapsto i,\ e\rangle_{\mathsf{env}}\ c\rangle_{\mathsf{cfg}} \wedge i \geq 0)$$

specifies the configurations where $\mathtt{x}$ is *bound to* a positive number. The following pattern states that $\mathtt{x}$ *points to* an existing location:

$$\begin{aligned}&\exists c{:}\mathit{Bag}_{\mathit{Cell}},\ e{:}\mathit{Env},\ p{:}\mathit{Nat},\ \ i{:}\mathit{Int},\ \sigma{:}\mathit{Map}_{\mathit{Nat,Int}}\\ &\quad\langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\end{aligned}$$

while the pattern

$$\exists c{:}\mathit{Bag}_{\mathit{Cell}},\ e{:}\mathit{Env},\ p{:}\mathit{Nat},\ \ i{:}\mathit{Int}\ \langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}$$

states also that the location $\mathtt{x}$ points to is the only one allocated.

Matching logic allows us to write specifications referring to data located arbitrarily deep in the configuration, at the same time allowing us to use existential variables to abstract away irrelevant parts of the configuration. To simplify writing, unless otherwise specified we adopt the following default notational conventions, mentioning that other configurations may benefit from other, or from more, similar notations.

Notation 1. *Variables starting with a "?" are assumed existentially quantified over the largest pattern of the formula containing them and thus need not be declared. Unless otherwise specified, the sorts of variables are inferred from their use context. Pattern existentially quantified variables which appear only once in the pattern can be replaced by an underscore (anonymous variable) "_" or by "...". Cells mentioned only for structural matching can be omitted when their presence is understood; e.g., if e is an environment and ψ a $\mathrm{FOL}_{=}$ formula, we may write $\langle e\rangle_{\mathsf{env}} \wedge \psi$ instead of $\langle\langle e\rangle_{\mathsf{env}}\ ...\rangle_{\mathsf{cfg}} \wedge \psi$.*

Only the last notation is specific to matching logic; the others are common in the literature and often tacitly used. With these notational conventions, the patterns above become:

$$\begin{aligned}&\langle \mathtt{x} \mapsto 5\ ...\rangle_{\mathsf{env}}\\ &\langle \mathtt{x} \mapsto ?i\ ...\rangle_{\mathsf{env}} \wedge ?i \geq 0\\ &\langle \mathtt{x} \mapsto ?p\ ...\rangle_{\mathsf{env}}\ \langle ?p \mapsto \_\ ...\rangle_{\mathsf{heap}}\\ &\langle \mathtt{x} \mapsto ?p\ ...\rangle_{\mathsf{env}}\ \langle ?p \mapsto \_\rangle_{\mathsf{heap}}\end{aligned}$$

Now that patterns can be written more compactly, let us further illustrate the expressiveness of matching logic by discussing a few

more examples. The following pattern states that program variables `x` and `y` are aliased and point to an existing location:

$$\langle \mathtt{x} \mapsto ?p,\ \mathtt{y} \mapsto ?p\ \ldots\rangle_{\mathsf{env}}\ \langle ?p \mapsto \_\ \ldots\rangle_{\mathsf{heap}}$$

The following patterns specify configurations in which the program variable `x` is bound to the last integer that has been output (the most recently output elements are to the right of the output cell), and configurations in which only one integer has been output and no program variable is bound to that integer, respectively:

$$\langle \mathtt{x} \mapsto ?i\ \ldots\rangle_{\mathsf{env}}\ \langle \ldots\ ?i\rangle_{\mathsf{out}}$$
$$\langle e\rangle_{\mathsf{env}}\ \langle ?i\rangle_{\mathsf{out}} \wedge ?i \notin \mathit{Codom}(e)$$

The following pattern states that the current function is `f` and that it has been called directly by `g` (stack's top is to the left):

$$\langle \mathtt{f}\rangle_{\mathsf{fname}}\ \langle(\mathtt{g}, \_, \_, \_)\ \ldots\rangle_{\mathsf{stack}}$$

The following pattern is more complex:

$$\langle \mathtt{x} \mapsto ?p\ \ldots\rangle_{\mathsf{env}}\ \langle \mathtt{f}\rangle_{\mathsf{fname}}\ \langle \ldots\ (\mathtt{g}, \_, \mathtt{x} \mapsto ?p\ \ldots, \mathtt{x} \mapsto \_^*\ \ldots)\ \ldots\rangle_{\mathsf{stack}}$$

It states that the current function is `f`, that it has been called directly or indirectly by function `g`, and that when `g` was stacked the program variable `x` had a pointer type and was bound to the same location ($?p$) to which it is also bound now in `f`'s environment.

3.3.2 Semantics

Informally, satisfaction in matching logic is defined in terms of *(pattern) matching* within the configuration model $\mathcal{T}$. More precisely, the satisfaction of the FOL$_=$ constructs is standard, while the satisfaction of basic patterns is defined by matching within $\mathcal{T}$. This intuition works best when $\mathcal{T}$ is thought of as an initial algebra (as it is in our MatchC instance of it), or at least as a term model, whose elements of sort *Cfg* are actually ground terms of sort *Cfg*. We first give matching logic a direct semantics and then we show that it reduces to conventional FOL$_=$ satisfaction in the fixed configuration model $\mathcal{T}$. Recall (see Section 3.1) that FOL$_=$ satisfaction in model $\mathcal{T}$ is defined in terms of $\mathcal{T}$-valuations and their associated satisfaction relation, that is, in terms of functions $\rho : \mathit{Var} \to \mathcal{T}$ and of a relation $\rho \models \psi$. Since the syntax of matching logic extends that of FOL$_=$ with basic patterns, which need their semantic counterpart, we extend FOL$_=$'s valuations to also include a configuration of $\mathcal{T}$, to be used for matching the basic patterns in the formula.

Definition 2. *We define the relation $(\gamma, \rho) \models \varphi$ over configurations $\gamma \in \mathcal{T}_{Cfg}$, valuations $\rho : \mathit{Var} \to \mathcal{T}$ and matching logic formulae φ as follows (for completeness, we also consider the FOL$_=$ constructs):*

$$\begin{array}{lll}
(\gamma,\rho) \models t = t' & \textit{iff} & \rho(t) = \rho(t') \\
(\gamma,\rho) \models \forall X\ \varphi & \textit{iff} & (\gamma,\rho') \models \varphi \textit{ for all } \rho' : \mathit{Var} \to \mathcal{T} \textit{ with } \rho'(y) = \rho(y) \textit{ for all } y \in \mathit{Var}\backslash X \\
(\gamma,\rho) \models \varphi \wedge \varphi' & \textit{iff} & (\gamma,\rho) \models \varphi \textit{ and } (\gamma,\rho) \models \varphi' \\
(\gamma,\rho) \models \neg\varphi & \textit{iff} & (\gamma,\rho) \not\models \varphi \\
(\gamma,\rho) \models \pi & \textit{iff} & \gamma = \rho(\pi)\ , \textit{ where } \pi \in \mathcal{T}_{\Sigma,Cfg}(\mathit{Var})
\end{array}$$

We write $\gamma \models \varphi$ whenever $(\gamma,\rho) \models \varphi$ for all $\rho : \mathit{Var} \to \mathcal{T}$, write $\rho \models \varphi$ whenever $(\gamma,\rho) \models \varphi$ for all $\gamma \in \mathcal{T}_{Cfg}$, and write $\models \varphi$ whenever $(\gamma,\rho) \models \varphi$ for all $\gamma \in \mathcal{T}_{Cfg}$ and all $\rho : \mathit{Var} \to \mathcal{T}$.

Assuming that $\mathcal{T}$ is a (ground) term model and γ is a configuration of the form $\langle\langle \mathtt{x} \mapsto 5,\ \mathtt{y} \mapsto 5\rangle_{\mathsf{env}}\ \langle 5 \mapsto 7\rangle_{\mathsf{heap}}\ \langle 3, 5\rangle_{\mathsf{out}}\ \ldots\rangle_{\mathsf{cfg}}$ where "..." stands for irrelevant parts of the configuration, then γ satisfies all the following patterns:

$$\begin{array}{lcl}
\pi_1 & \equiv & \langle \mathtt{x} \mapsto 5\ \ldots\rangle_{\mathsf{env}} \\
\pi_2 & \equiv & \langle \mathtt{x} \mapsto ?i\ \ldots\rangle_{\mathsf{env}} \wedge ?i \geq 0 \\
\pi_3 & \equiv & \langle \mathtt{x} \mapsto ?p\ \ldots\rangle_{\mathsf{env}}\ \langle ?p \mapsto _\ \ldots\rangle_{\mathsf{heap}} \\
\pi_4 & \equiv & \langle \mathtt{x} \mapsto ?p\ \ldots\rangle_{\mathsf{env}}\ \langle ?p \mapsto _\rangle_{\mathsf{heap}} \\
\pi_5 & \equiv & \langle \mathtt{x} \mapsto ?p,\ \mathtt{y} \mapsto ?p\ \ldots\rangle_{\mathsf{env}}\ \langle ?p \mapsto _\ \ldots\rangle_{\mathsf{heap}} \\
\pi_6 & \equiv & \langle \mathtt{x} \mapsto ?i\ \ldots\rangle_{\mathsf{env}}\ \langle \ldots\ ?i\rangle_{\mathsf{out}}
\end{array}$$

Moreover, $\models \pi_1 \to \pi_2$, $\models \pi_3 \to \pi_2$, $\models \pi_4 \to \pi_3$, $\models \pi_5 \to \pi_3$, and, assuming that $\mathcal{T}$ correctly defines the claimed maps, lists, etc., $\models \pi_1 \wedge \pi_5 \wedge \pi_6 \to \langle \mathtt{y} \mapsto 5\ \ldots\rangle_{\mathsf{env}}\ \langle 5 \mapsto \_\ \ldots\rangle_{\mathsf{heap}}\ \langle \ldots\ 5\rangle_{\mathsf{out}}$.

We next show how matching logic formulae can be translated into FOL$_=$ formulae, so that its satisfaction becomes FOL$_=$ satisfaction in the model of configurations, $\mathcal{T}$.

Definition 3. *Let $\square$ be a fresh variable of sort Cfg. If φ is a matching logic formula, then let $\varphi^{\square}$ be the FOL$_=$ formula replacing each basic pattern $\pi \in \mathcal{T}_{\Sigma,Cfg}(\mathit{Var})$ in φ by the equality $\square = \pi$. If $\rho : \mathit{Var} \to \mathcal{T}$ and $\gamma \in \mathcal{T}_{Cfg}$ then let $\rho^{\gamma} : \mathit{Var} \cup \{\square\} \to \mathcal{T}$ be the mapping defined as $\rho^{\gamma}(x) = \rho(x)$ for all $x \in \mathit{Var}$ and $\rho^{\gamma}(\square) = \gamma$.*

Proposition 1. *If φ is a matching logic formula, $\rho : \mathit{Var} \to \mathcal{T}$ and $\gamma \in \mathcal{T}_{Cfg}$, then $(\gamma,\rho) \models \varphi$ iff $\rho^{\gamma} \models_{\mathrm{FOL}_=} \varphi^{\square}$. Also, $\models \varphi$ iff $\mathcal{T} \models_{\mathrm{FOL}_=} \varphi^{\square}$.*

Proposition 1 thus tells us that matching logic can be framed as a methodological fragment of FOL$_=$ for a particular model. This fact may appear disappointing to some readers, so it deserve some explanations. First, it is actually a strong point in matching logic's favor. Indeed, in spite of its descriptive power, we can actually use conventional theorem provers for matching logic reasoning, provided that we have a good FOL$_=$ formalization of the configuration model $\mathcal{T}$; the performance and complexity of our experiments with the MatchC verifier (see Section 5) may serve as an argument in this direction. Second, one should not under-estimate the expressive power of particular models, particularly initial models. While FOL$_=$ is complete and thus its satisfaction problem is semi-decidable, the satisfaction (of FOL$_=$ sentences) problem in particular models can be anywhere in the arithmetic hierarchy (e.g., the equational satisfaction problem in initial models is Π^0_2-complete).

The above being said, we have found that it is hard to work with a model in practice. Instead, in our implementation, we simply identify $\mathcal{T}$ with the properties that we have accumulated for it in our background theory, i.e., properties about lists, sets, bags, maps, etc. (e.g., the associativity and commutativity of constructs for sets, bags, maps, uniqueness of bindings in maps, etc.). From here on in this paper we take a similar approach, but more theoretical.

Definition 4. *Let $\mathcal{T}^* = \{\psi \mid \mathcal{T} \models_{\mathrm{FOL}_=} \psi\}$ be the FOL$_=$ theory of $\mathcal{T}$.*

$\mathcal{T}^*$ is not expected to be decidable or semi-decidable in general. In practice, however, we typically work with a convenient subset of it, say $F \subset \mathcal{T}^*$. How such a subset F can be found is interesting, challenging and very useful, but we are not concerned with it here. Then Proposition 1 essentially reduces, in practice, the problem of matching logic reasoning to ordinary FOL$_=$ reasoning: $\models \varphi$ in matching logic if $F \models_{\mathrm{FOL}_=} \varphi^{\square}$. Notice that one is free to manually prove theorems of $\mathcal{T}^*$, possibly with the help of a proof assistant.

3.4 Abstractions

Since matching logic can be framed within FOL$_=$, conventional abstraction mechanisms can be used. In this section, however, we propose an abstraction approach motivated by particularities of matching logic, which turned out to be quite effective in practical experiments with MatchC. The basic idea is to introduce and axiomatize situations of interest as *operations* rather than as predicates.

Suppose that one is only interested in the fact that two program variables are aliased and their location is alive, but what is the particular location to which they point is irrelevant. Then we can add to Σ an abstraction operation for configurations together with an axiom for it as follows (we quantify all variables and give their types to avoid notational confusion):

$$\mathit{Cfg} ::= \mathsf{aliased}(\mathit{Id}, \mathit{Id})$$

$$\forall x{:}\mathit{Id},\ y{:}\mathit{Id} \quad \mathsf{aliased}(x, y) \leftrightarrow \exists p{:}\mathit{Nat},\ \rho{:}\mathit{Env},\ i{:}\mathit{Int},\ \sigma{:}\mathit{Map}_{\mathit{Nat,Int}},\ c{:}\mathit{Bag}_{\mathit{Cell}}\ \langle\langle x \mapsto p,\ y \mapsto p,\ \rho\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}$$

We can now use aliased in matching logic specifications and reasoning. For example, implications like the one below can be easily proved (using $\text{FOL}_{=}$; "..." stands for irrelevant subterms):

$$\models \quad \langle\langle \mathbf{x} \mapsto 3,\ \mathbf{y} \mapsto 3\ ...\rangle_{\mathsf{env}}\ \langle 3 \mapsto 5\ ...\rangle_{\mathsf{heap}}\ ...\rangle_{\mathsf{cfg}} \rightarrow \mathsf{aliased}(\mathbf{x}, \mathbf{y})$$

We next show the list heap abstraction which is part of the library of MatchC and which was used, together with other similar abstractions, to verify the programs in Section 2. It abstracts heap subterms into list terms and captures two cases, one in which the list is empty and the other in which it has at least one element. We borrowed the notation $p \mapsto [a, q]$ from separation logic, but in our case it stands for a two-binding heap subterm, "$p \mapsto a,\ p+1 \mapsto q$".

$$\begin{array}{ll} & \langle\langle \mathsf{list}(p)(\alpha), \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \\ \leftrightarrow & \langle\langle \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ p = 0\ \wedge\ \alpha = [] \\ \vee & \exists a, q, \beta\ (\langle\langle p \mapsto [a, q], \mathsf{list}(q)(\beta), \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ \alpha = [a]@\beta) \end{array}$$

One can now use this axiom to perform $\text{FOL}_{=}$ reasoning like below:

$$\begin{array}{ll} & \langle\langle 1 \mapsto 5,\ 2 \mapsto 0,\ 7 \mapsto 9,\ 8 \mapsto 1,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{config}} \\ \leftrightarrow & \langle\langle 1 \mapsto 5,\ 2 \mapsto 0,\ \mathsf{list}(0)([]),\ 7 \mapsto 9,\ 8 \mapsto 1,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{config}} \\ \leftrightarrow & \langle\langle \mathsf{list}(1)([5]),\ 7 \mapsto 9,\ 8 \mapsto 1,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{config}} \\ \rightarrow & \langle\langle \mathsf{list}(7)([9, 5]),\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{config}} \\ \leftrightarrow & \exists q\ \langle\langle 7 \mapsto 9,\ 8 \mapsto q,\ q \mapsto 5,\ q+1 \mapsto 0,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{config}} \end{array}$$

Note that by defining configuration abstractions as operations instead of predicates, we avoid having to define recursive predicates in matching logic. The axiom above simply constrains terms and thus is like any other FOL axiom, using no new predicates or logical connectives. However, note that we can still not avoid difficult inconsistency aspects due to wrong abstractions, that is, we still have to check that for a given set of axioms $\mathcal{F}$, it is *not* the case that $\mathcal{T}^* \cup \mathcal{F} \models \mathit{false}$. We currently do not check consistency in MatchC.

4. Matching Logic Rewriting

Matching logic has been introduced in [47] as a vehicle to give better axiomatic semantics. The main result of [47] says that with matching logic one can give axiomatic semantics which are forwards (like the Floyd rule for assignment) and do not introduce new quantifiers (like the Hoare rule for assignment); moreover, the resulting semantics is equivalent to Hoare logic, so it shares the same good properties (modularity, partial correctness, relative completeness). Unfortunately, the approach in [47] shares a major disadvantage with other axiomatic approaches: the target language needs to be given a new, axiomatic semantics. Axiomatic semantics are less intuitive than operational semantics, are not easily executable, and are hard to test. What we want is *one* formal semantics of a programming language, which should be both executable and suitable for program verification. This is what we do in this section.

4.1 Matching Logic Rewriting and Language Semantics

Operational semantics are typically given in terms of program configurations. Since matching logic has configurations at its core, it has been used in a straightforward manner to define operational semantics by leveraging the wealth of techniques, methodologies and tools developed by the term rewriting and reduction semantics communities. In this section we show how the basic notion of a rewrite (or reduction) rule is smoothly captured by a more general notion of rewriting, between matching logic formulae. Before that, we first motivate our choice.

As mentioned in Section 1, there are various tool-supported operational semantics approaches in which a language is defined as a set of rewrite or reduction rules "$l \Rightarrow r$ if b", where l and r are configuration terms with variables constrained by boolean condition b. One of the most popular approaches is reduction semantics with evaluation contexts [18, 19], with rules "$c[t] \Rightarrow c[t']$ if b", where c is an evaluation context, t is the redex which reduces to t', and b a side condition. Another approach is the chemical abstract machine [8], where l is a chemical solution that reacts into r under condition b. The rewriting logic semantics framework $\mathbb{K}$ [44] is yet another approach, based on plain (no evaluation contexts) rewrite rules of the form "$l \Rightarrow r$ if b". Finally, higher-order logic is also a successful framework for defining operational semantics [9, 26, 50], and it is technically the most powerful of all the above, in that any operational semantic approach can also be done in higher-order logic.

Any of the above can be used for defining operational semantics based on rewrite rules of the form "$l \Rightarrow r$ if b". In our current implementation (see Section 5), we picked the rewriting logic semantics approach for a series of (admittedly subjective) reasons:

- First, rewriting logic gives us symbolic execution at no additional effort, in that rewrite rules which were conceived to work with concrete domain values apply to symbolic values as well; e.g., adding a constant n of sort *Nat* gives us symbolic natural numbers thanks to its initial model semantics, and rules like $(\forall x)\ 0 * x \Rightarrow 0$ apply also to symbolic terms like $0*(n+1)$ rewriting them to 0. This symbolic nature of rewriting logic facilitates significantly the implementation of matching logic verifiers.
- Second, rewriting logic has good tool support. Maude [12] is a high-performance rewriting logic engine, with support for efficient execution of rewrite systems, for exhaustive state-space analyses (including an LTL model checker), as well as for consistency analysis of rewrite systems. The use of Maude was a critical factor in the development of MatchC, both w.r.t. speed of development and w.r.t. performance of verification.
- Third, it is the simplest logical framework among the above, requiring no evaluation contexts (whose semantics and implementation are nontrivial [29]) or expensive "airlock" operations [8]. There is no doubt that rewriting logic semantics can be embedded in higher-order logic, as rewriting is so basic, while it is not clear that one can go the other way around.
- Finally, a large community of researchers are actively involved in the *rewriting logic semantics project* [36], whose aim is to use rewriting logic and Maude as a framework for programming language semantics. One of the results of this project is $\mathbb{K}$ [44], which we use in our implementation and briefly discuss next.

Let us briefly discuss how we can define operational semantics in $\mathbb{K}$ ultimately (after its syntactic desugaring) using only rewrite rules of the form "$l \Rightarrow r$ if b", where $l, r \in T_{\Sigma, Cfg}(X)$ are configuration terms with variables in set X and b is a boolean condition only constraining the variables in l and r but involving no rewrite rule premises. $\mathbb{K}$ borrows from abstract state machines and from techniques like refocusing [14] the idea of flattening syntax into a list/stack of computational tasks, thus allowing to systematically pull out from context parts that need to be evaluated and plug back into context their results. For example, the evaluation strategy of the conditional statement in our fragment of C is given by the following pair of complementary $\mathbb{K}$ rules:

$$\begin{array}{l} \langle\langle \texttt{if(}e\texttt{)}\ s\ \texttt{else}\ s' \Rightarrow e \curvearrowright \texttt{if(}\blacksquare\texttt{)}\ s\ \texttt{else}\ s'\ ...\rangle_{\mathsf{k}}\ ...\rangle_{\mathsf{cfg}}\ \text{ if } e \notin \mathit{Int} \\ \langle\langle e \curvearrowright \texttt{if(}\blacksquare\texttt{)}\ s\ \texttt{else}\ s' \Rightarrow \texttt{if(}e\texttt{)}\ s\ \texttt{else}\ s'\ ...\rangle_{\mathsf{k}}\ ...\rangle_{\mathsf{cfg}}\ \text{ if } e \in \mathit{Int} \end{array}$$

The "$\blacksquare$" is a new syntactic constant with the meaning "plug here" and "$\curvearrowright$" is a special syntactic list construct with the meaning "plugged into". $\mathbb{K}$ provides a series of notations allowing quite compact rewriting logic semantic definitions. For example, the two rules above are automatically generated from a compact "`strict(1)`" attribute associated to the conditional, stating that it is strict in its first argument. Perhaps the most important notation of $\mathbb{K}$, which we also adopted in MatchC and can be seen in the examples in Section 2, is its in-place rewriting: $cxt[t_1 \Rightarrow t'_1, ..., t_n \Rightarrow t'_n]$ instead of $cxt[t_1, ..., t_n] \Rightarrow cxt[t'_1, ..., t'_n]$. This allows not only for a

more compact and less error-prone notation since one needs not repeat the identical context *cxt* (which can be large) in both sides of the rule, but more importantly it highlights the *structural frame*, that is, the parts of the term which do not change. Notation 1 is taken over from $\mathbb{K}$, which uses "..." for structural framing.

Rules like above eventually bring the redex to the top of the $\langle ... \rangle_{\mathsf{k}}$ cell, which can then be matched and rewritten with rules like

$\langle\langle\ x \Rightarrow i\ ... \rangle_{\mathsf{k}}\ \langle x \mapsto i\ ... \rangle_{\mathsf{env}}\ ... \rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{if(0)}\ \_\ \texttt{else}\ s \Rightarrow s\ ... \rangle_{\mathsf{k}}\ ... \rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{if(}i\texttt{)}\ s\ \texttt{else}\ \_ \Rightarrow s\ ... \rangle_{\mathsf{k}}\ ... \rangle_{\mathsf{cfg}}$ if $i \neq 0$
$\langle\langle \texttt{while(}e\texttt{)}\ s \Rightarrow \texttt{if(}e\texttt{)}\ \{\ s\texttt{;}\ \texttt{while(}e\texttt{)}\ s\ \}\ \texttt{else}\ \{\}\ ... \rangle_{\mathsf{k}}\ ... \rangle_{\mathsf{cfg}}$

The first rule above gives the semantics of variable lookup, the next two the semantics of the conditional, and the fourth the unrolling semantics of the while loop. The $\mathbb{K}$ semantics of the C fragment considered so far in MatchC consists of 41 syntactic constructs (Σ) and 91 rules like the ones above, excluding the automatically generated ones corresponding to strictness attributes; most of these are discussed in [46]. Due to its cell-based configurations and its rewriting and framing philosophy, each rule matching only what it needs, $\mathbb{K}$ scales well. Several complete languages have been given formal executable $\mathbb{K}$ semantics so far, including an almost complete semantics of C [17] (it passes 99.2% of GCC's implementation-independent torture test suite; for a comparison, GCC itself only passes 99%, ICC passes 99.4%, and Clang 98.3%; see `http://c-semantics.googlecode.com`). To avoid discussing $\mathbb{K}$ in depth here (see [44] and `http://k-framework.org`), we wrote the rules above more verbosely than needed in $\mathbb{K}$.

To summarize the discussion above, there is enough practical evidence that the fragment of rewriting logic consisting of only rules $(\forall X)\ l \Rightarrow r$ if b, where $l, r \in T_{\Sigma,Cfg}(X)$ and b is a boolean condition involving no rewrite rule premises, is powerful enough to support realistic operational semantics. Consequently, we currently focus on capturing only this fragment in our matching logic setting.

Definition 5. *A (matching logic)* ***rewrite rule*** *is a pair $\varphi \Rightarrow \varphi'$, where φ and φ' are matching logic formulae (not necessarily closed). A (matching logic)* ***rewrite system*** *is a set of rewrite rules.*

As usual with rewrite rules, we call φ the left-hand side (LHS) and φ' the right-hand side (RHS) of the rule $\varphi \Rightarrow \varphi'$. Note that conventional rewrite rules over terms of sort *Cfg* are just special matching logic rewrite rules. Indeed, a *Cfg* unconditional rewrite rule $(\forall X)\ l \Rightarrow r$ is nothing but a matching logic rewrite rule $\varphi \Rightarrow \varphi'$ where φ and φ' are the basic patterns l and r, respectively, with the variables in X left free in both φ and φ'. The same holds true for conditional rules with boolean conditions: a conditional rewrite rule "$(\forall X)\ l \Rightarrow r$ if b" is equivalent to any of the matching logic rewrite rules over constrained patterns $l \wedge b \Rightarrow r$ or $l \wedge b \Rightarrow r \wedge b$. Therefore, an operational semantics defined using any of the approaches above can be regarded as a matching logic rewrite system.

From now on, by a rewrite rule we mean a matching logic rewrite rule and by a rewrite system we mean a matching logic rewrite system. As a notational convenience, we write $\mathcal{A}$ when we mean a generic rewrite system, and $\mathcal{S}$ when we mean a rewrite system corresponding to the semantics of a programing language.

Definition 6. *A rewrite system $\mathcal{S}$ induces a* ***transition system*** *$(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ on the configuration model: $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ for some $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is some rule $\varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and some $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$. Configuration $\gamma \in \mathcal{T}_{Cfg}$* ***terminates*** *in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ iff there is no infinite $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-sequence $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ starting with γ. A rewrite rule $\varphi \Rightarrow \varphi'$ is* ***well-defined*** *iff for any $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$, there is some $\gamma' \in \mathcal{T}_{Cfg}$ with $(\gamma', \rho) \models \varphi'$. Rewrite system $\mathcal{S}$ is* ***well-defined*** *iff each rule of it is well-defined, and is* ***deterministic*** *iff $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is deterministic.*

Based on the discussion above, it is expected that any programming language is given operational semantics as a rewrite system $\mathcal{S}$, which generates a transition system on the configuration model, $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, telling precisely how the language in question operates. Note that well-definedness does not come for granted: e.g., a rule $l \Rightarrow$ *false* is not well-defined. Nevertheless, rewrite systems corresponding to operational semantics like above contain only rules $l \wedge b \Rightarrow r$ with l and r basic patterns, which are well-defined.

We next define semantic validity in matching logic rewriting. Recall from Section 1 that a Hoare triple $\{\psi\}\ \texttt{code}\ \{\psi'\}$ can be regarded as a rewrite rule whose left-hand-side embeds `code` in its configuration and adds ψ as additional constraints, and whose right-hand-side contains an empty code in its configuration and only adds ψ'. In conventional axiomatic semantics, a (partial correctness) Hoare triple is *semantically valid*, written $\models \{\psi\}\ \texttt{code}\ \{\psi'\}$, iff for any state $s \models \psi$, if `code` executed in state s terminates with state s' then $s' \models \psi'$. This elegant definition of semantic validity has the luxury of relying on another formal semantics of the language, which provides the notions of "execution", "termination", and "state". Since in matching logic rewriting all these happen in the same semantics which is given as a matching logic rewrite system, and since the closest matching logic element to a "state" is a ground configuration in $\mathcal{T}_{Cfg}$ including both the state and the code, and since the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ gives all the operational behaviors of the defined language, we introduce the following:

Definition 7. *Let $\mathcal{S}$ be a rewrite system and $\varphi \Rightarrow \varphi'$ a rewrite rule. Then $\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff for all $\gamma \in \mathcal{T}_{Cfg}$ such that γ terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ and for all $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$, there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$.*

The matching logic rewrite rules are more expressive than the Hoare triples, since φ' needs not have an empty code cell as implicitly assumed in Hoare triples. If φ' has an empty code cell then so does γ' in the definition above, and we get the Hoare validity.

4.2 Matching Logic Operational Deduction

Let us first consider only the first eight proof rules in Figure 1. We here show that these language-independent rules are sound and complete for the operational behaviors of the language defined by $\mathcal{S}$, that is, for $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$. That means that these rules can be safely used to derive correct computations within the operational semantics (soundness), and that any correct computation using the operational semantics can also be derived using these rules (completeness). Only the first four rules are needed for completeness; the role of the others becomes more visible in Section 4.3. For the reminder of the paper, whenever we write "$\mathcal{A} \vdash \varphi \Rightarrow^{1\text{-}8} \varphi'$" we mean "$\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is derivable with the first eight rules of the proof system in Figure 1", and when we write "$\mathcal{A} \vdash \varphi \Rightarrow^{1\text{-}4} \varphi'$" we mean "$\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is derivable with the first four proof rules.

Before formalizing and proving the operational soundness and completeness of these eight rules, let us first discuss an example.

Example 1. Consider the **`while`** loop in function `reverseList` discussed earlier (Figure 3), modified to only iterate at most once, that is, modified into a conditional, and let us show that it satisfies the claimed invariant. More precisely, let us prove the rewrite rule in Figure 8, say $\exists X \varphi \Rightarrow \exists X \varphi'$, where $X = \{\texttt{?x}, \texttt{?p}, \texttt{?p}, \texttt{?B}, \texttt{?C}\}$. Recall that, by convention, the "?" variables are existentially quantified over their corresponding patterns. The matching logic rule in Figure 8 looks different from the invariant in Figure 3 because of two MatchC notations: first, MatchC desugars invariants `inv` φ `loop` into matching logic rules $\varphi[\texttt{loop...}] \Rightarrow \varphi[...] \wedge \neg\, cond(\texttt{loop})$ (see Section 5) , where $\varphi[\texttt{code}]$ is the pattern obtained from φ by making the contents of its $\langle ... \rangle_{\mathsf{k}}$ cell `code`; second, as explained in Section 1, MatchC allows the user to refer directly

$$
\begin{array}{l}
\langle \\
\quad \langle \cdots\ \texttt{x} \mapsto ?\texttt{x},\ \texttt{p} \mapsto ?\texttt{p},\ \texttt{y} \mapsto \texttt{y}\ \cdots \rangle_{\mathsf{env}} \\
\quad \langle \cdots\ \mathsf{list}(?\texttt{p})(?\mathsf{B}),\ \mathsf{list}(?\texttt{x})(?\mathsf{C})\ \cdots \rangle_{\mathsf{heap}} \\
\quad \langle\ \texttt{if(x != NULL) \{} \\
\qquad \texttt{y = x->next;} \\
\qquad \texttt{x->next = p;} \\
\qquad \texttt{p = x;} \\
\qquad \texttt{x = y;} \\
\quad \texttt{\}}\ \cdots \rangle_{\mathsf{k}} \\
\quad \ldots \\
\rangle_{\mathsf{cfg}}\ \wedge\ \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C} \\
\Rightarrow \\
\langle \\
\quad \langle \cdots\ \texttt{x} \mapsto ?\texttt{x},\ \texttt{p} \mapsto ?\texttt{p},\ \texttt{y} \mapsto \texttt{y}\ \cdots \rangle_{\mathsf{env}} \\
\quad \langle \cdots\ \mathsf{list}(?\texttt{p})(?\mathsf{B}),\ \mathsf{list}(?\texttt{x})(?\mathsf{C})\ \cdots \rangle_{\mathsf{heap}} \\
\quad \langle \cdots \rangle_{\mathsf{k}} \\
\quad \ldots \\
\rangle_{\mathsf{cfg}}\ \wedge\ \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C}
\end{array}
$$

Figure 8. Matching logic rewrite rule derivable with the first eight rules of the proof system in Figure 1 with $\mathcal{S}$ the executable semantics of the considered fragment of C (see Section 4.1 and [46]). The ellipses in each pattern stand for distinct free variables, assumed the same on the corresponding positions in the LHS and the RHS. The "?" variables are existentially quantified over each pattern.

to program variable `x` instead of logical variable ?x, generating automatically the environment cell containing bindings of the form $\texttt{x} \mapsto ?\texttt{x}$. Since here we want to illustrate a formal proof, we completely desugar the MatchC notation in Figure 3. We first derive $\varphi \Rightarrow \exists X\varphi'$ and then the desired rule follows by **Abstraction**. To derive $\varphi \Rightarrow \exists X\varphi'$, it suffices to derive $\varphi \wedge ?\mathsf{x} = 0 \Rightarrow \exists X\varphi'$ and $\varphi \wedge ?\mathsf{x} \neq 0 \Rightarrow \exists X\varphi'$, since we can then use **Case analysis** and **Consequence**:

- For the former, we iteratively use the executable semantic rules in $\mathcal{S}$ of the considered fragment of C via **Axiom**, **Substitution** and **Logical Framing**, together with $\mathrm{FOL}_{=}$ reasoning via **Consequence** and with the **Transitivity** rule, until the condition of `if` evaluates to 0 and then the `if` statement dissolves (its else branch is empty), thus obtaining $\varphi \wedge ?\mathsf{x} = 0 \Rightarrow \varphi' \wedge ?\mathsf{x} = 0$. The derivation of $\varphi \wedge ?\mathsf{x} = 0 \Rightarrow \exists X\varphi'$ follows via **Consequence**, since $\models \varphi' \wedge ?\mathsf{x} = 0 \rightarrow \exists X\varphi'$.
- For the latter, like above we also use **Axiom**, **Substitution**, **Logical Framing**, **Consequence** and **Transitivity** until the condition of `if` evaluates to $?\mathsf{x} \neq 0$, then we apply the semantics of `if` and take the then branch. To continue with the execution of the other statements, we need to apply the list axiom in Section 3.4 from left-to-right; $\mathrm{FOL}_{=}$ reasoning eliminates the case when the list is empty (since $?\mathsf{x} \neq 0$). Then **Abstraction** allows us to assume fresh variables a, q and β like in the axiom of lists in Section 3.4 and thus we can continue with the execution. After the block terminates, we have:

$$
\begin{array}{l}
\langle \\
\quad \langle \cdots\ \texttt{x} \mapsto q,\ \texttt{p} \mapsto ?\texttt{x},\ \texttt{y} \mapsto q\ \cdots \rangle_{\mathsf{env}} \\
\quad \langle \cdots\ \mathsf{list}(?\texttt{p})(?\mathsf{B}), ?\texttt{x} \mapsto [a, ?\texttt{p}],\ \mathsf{list}(q)(\beta)\ \cdots \rangle_{\mathsf{heap}} \\
\quad \langle \cdots \rangle_{\mathsf{k}} \\
\quad \ldots \\
\rangle_{\mathsf{cfg}}\ \wedge\ ?\mathsf{C} = [a]@\beta\ \wedge\ \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C}
\end{array}
$$

  Let φ'' denote this pattern. We can now again use $\mathrm{FOL}_{=}$ reasoning, this time applying the list axiom from right-to-left and using properties of the configuration model $\mathcal{T}$ (like those in Section 3.2), and eventually derive $\varphi'' \Rightarrow \exists X\varphi'$.

If we had not modified the `while` loop into an `if` conditional in the φ pattern above, then the second case above would have started by first applying the rewriting semantics of the `while` loop (see Section 4.1), namely unrolling into an `if`, and then the proof would have followed similarly until a pattern like the φ'' above was reached, but one where the $\langle ... \rangle_{\mathsf{k}}$ cell contains the original `while` loop. Next one can either continue to unroll the loop or one can conclude, similarly to the above, that $\varphi'' \Rightarrow \exists X\varphi$. Unfortunately, none of these would prove the original goal, $\exists X\varphi \Rightarrow \exists X\varphi'$.

We will show how the last rule of the proof system in Figure 1 can be used to deal with circular behaviors in Section 4.3.

In the reminder of this section we prove the soundness and completeness of the first eight rules for deriving operational behaviors.

Proposition 2. ***(operational soundness)*** *If $\mathcal{S}$ is well-defined, $\mathcal{S} \vdash \varphi \Rightarrow^{1\text{-}8} \varphi'$, and $(\gamma, \rho) \models \varphi$ for some $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$, then there is some $\gamma' \in \mathcal{T}_{Cfg}$ with $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$.*

The following simple result says that if the configuration model is chosen to be a term model, then transitions in it can also be mimicked with the proof system (only the first four rules are needed).

Proposition 3. ***(operational completeness)*** *If all the rewrite rules in $\mathcal{S}$ only use conjunctive patterns, $\mathcal{T}$ is chosen to be a term model, and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ for some $\gamma, \gamma' \in \mathcal{T}_{Cfg}$, then $\mathcal{S} \vdash \gamma \Rightarrow^{1\text{-}4} \gamma'$.*

As seen in Section 4.1, rewrite systems corresponding to rewrite-based operational semantics only use conjunctive patterns. We need this hypothesis to guarantee that $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ can be derived using the **Substitution** rule with ground substitutions $\theta : X \to \mathcal{T}_{\Sigma}$.

Theorem 1. ***(operational soundness and completeness)*** *If $\mathcal{S}$ is well-defined, its rules only use conjunctive patterns, and $\mathcal{T}$ is a term model, then $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ iff $\mathcal{S} \vdash \gamma \Rightarrow^{1\text{-}8} \gamma'$ for any $\gamma, \gamma' \in \mathcal{T}_{Cfg}$.*

4.3 Matching Logic Program Verification

The results in Section 4.2 tell us that, under reasonable conditions, the first eight rules of the proof system in Figure 1 faithfully capture the operational semantics of the language defined using a set of rewrite rules $\mathcal{S}$. Moreover, they can also be used to derive some rewriting properties of the language. However, one cannot expect them to be too powerful as a program verification foundation, because, as seen in the Example in Section 4.2, they cannot cope with circular behaviors. In this section we show how the ninth rule in Figure 1, **Circularity**, can deal with circular behaviors.

Definition 8. *Let $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'$ be the derivation relation obtained by dropping the* **Reflexivity** *rule from the proof system in Figure 1.*

The intuition for $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'$ is that a configuration satisfying φ needs at least one semantic step to transit to one satisfying φ'. All results in Section 4.2 also hold when we replace $\Rightarrow$ by $\Rightarrow^{+}$ and $\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$ by $\Rightarrow_{\mathcal{S}}^{+\mathcal{T}}$. Let us now consider the **Circularity** rule in Figure 1. It says that we can derive the sequent $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ whenever we can derive the rule $\varphi \Rightarrow \varphi'$ by starting with one or more rewrite steps in $\mathcal{A}$ and continuing with steps which can involve both rules from $\mathcal{A}$ and the rule to be proved itself, $\varphi \Rightarrow \varphi'$. The first step can for example be a loop unrolling step in the case of loops, or a function invocation step in the case of recursive functions, etc.

Example 2. We show how the **Circularity** rule can be used to verify the **`while`** loop in function `reverseList` in Figure 3. In Example 1, we showed how the first eight rules of the proof system in Figure 1 can be used to verify that the claimed loop invariant holds after the execution of the code obtained by modifying the **`while`** loop into an **`if`** conditional, and we argued that they cannot derive the desired property about the **`while`** loop, essentially

$$\textbf{Set circularity:}\quad \frac{\begin{array}{c} C \text{ is the set } \{\ \varphi_1 \Rightarrow \varphi'_1,\ \ldots,\ \varphi_n \Rightarrow \varphi'_n\ \} \\ \mathcal{A} \ \vdash\ \{\ \varphi_1 \Rightarrow^+ \varphi''_1,\ \ldots,\ \varphi_n \Rightarrow^+ \varphi''_n\ \} \\ \mathcal{A} \cup C \ \vdash\ \{\ \varphi''_1 \Rightarrow \varphi'_1,\ \ldots,\ \varphi''_n \Rightarrow \varphi'_n\ \} \end{array}}{\mathcal{A} \ \vdash\ C}$$

Figure 9. Derived circularity rule schema

because of their lack of reasoning support for circular behaviors. Let $\exists X\varphi \Rightarrow \exists X\varphi'$ be the desired property, that is, the matching logic rewrite rule in Figure 8 with **if** modified into **while**. Since φ contains the **while** loop in its $\langle ... \rangle_k$ cell, we can use the loop unrolling rewrite rule in $\mathcal{S}$ and rewrite φ into a pattern that resembles the LHS of the rule in Figure 8, except that the `then` branch of the **if** is followed by the **while** loop. Let us call this pattern $\varphi_{\texttt{if}}$. Using **Consequence** and **Abstraction**, we can thus derive $\mathcal{A} \ \vdash\ \exists X\varphi \Rightarrow^+ \exists X\varphi_{\texttt{if}}$. Thanks to the **Circularity** rule, it suffices to derive $\mathcal{A} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \ \vdash\ \exists X\varphi_{\texttt{if}} \Rightarrow \exists X\varphi'$. Using a reasoning sequence similar to the one in Example 1, we can first derive $\varphi_{\texttt{if}} \wedge ?\texttt{x} = 0 \ \Rightarrow\ \exists X\varphi'$ and then $\varphi_{\texttt{if}} \wedge ?\texttt{x} \neq 0 \ \Rightarrow\ \exists X\varphi$. We can now use **Transitivity** and the rule in $\exists X\varphi \Rightarrow \exists X\varphi'$ to derive $\mathcal{A} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \ \vdash\ \varphi_{\texttt{if}} \wedge ?\texttt{x} \neq 0 \Rightarrow \exists X\varphi'$, then **Case analysis** to derive $\mathcal{A} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \ \vdash\ \varphi_{\texttt{if}} \Rightarrow \exists X\varphi'$, and then **Abstraction** to finally derive $\mathcal{A} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \ \vdash\ \exists X\varphi_{\texttt{if}} \Rightarrow \exists X\varphi'$. For a formal detailed proof see Section B of the Appendix.

The use of the claimed properties in their own proofs in the rule **Circularity** is reminiscent of *circular coinduction* [43]. However, like in circular coinduction, where the claimed properties can only be used in some special contexts, the **Circularity** rule also disallows their unrestricted use: it only allows them to be used *after* a step using the original semantics has been performed. We believe that our current restriction can be weaken a bit (e.g., by allowing a **Case analysis** as last step, instead of an implicit transitivity, where each case starts with its own rule in $\mathcal{A}$), but note that one cannot allow arbitrary use of the claimed rules in their proofs. For example, if $\mathcal{A}$ contains $\varphi_1 \Rightarrow \varphi_2$ then $\varphi_2 \Rightarrow \varphi_1$ can be "proved" in a two-step transitivity, using itself, the rule in $\mathcal{A}$ and then itself again. We therefore have to restrict the use of the rules of C in their own proofs. In all our experiments so far, it was always possible to give the candidate rules proofs which start with a rule in $\mathcal{A}$.

We next state our main result. If a language is deterministic then its transition system is deterministic (see Definition 6). Note, however, that program specifications tend to be non-deterministic in practice, because they typically over-approximate the program behavior. If a rewrite system is well-defined and deterministic, then each $\gamma \in \mathcal{T}_{Cfg}$ admits a unique rewrite sequence.

Theorem 2. ***(partial correctness)*** *Let $\mathcal{S}$ be a well-defined and deterministic set of rewrite rules, and $\mathcal{S} \ \vdash\ \varphi \Rightarrow \varphi'$ a sequent derived with the proof system in Figure 1. Then $\mathcal{S} \models \varphi \Rightarrow \varphi'$.*

The **Circularity** proof rule in Figure 1 allows only for circularly deriving one rewrite rule at a time. However, we sometimes want to circularly derive several rewrite rules at the same time. This happens, for example, when we verify mutually recursive functions or fragments of code. It turns out that our **Circularity** rule in Figure 1 is powerful enough to handle such mutually circular proofs. In what follows we first propose a more general rule deriving mutual circularities and then we show that it is unnecessary.

Figure 9 shows the **Set circularity** proof rule schema. Note that it uses sets of rewrite rules in the right-hand sides of the sequents; it actually is syntactic sugar for saying that each of the rules is derivable. That is, if $C = \{\ \varphi_1 \Rightarrow \varphi'_1, \ldots,\ \varphi_n \Rightarrow \varphi'_n\ \}$ is a finite set of rules and $\mathcal{A}$ is a rewrite system, then $\mathcal{A} \ \vdash\ C$ is a notation for the fact that each of the sequents $\mathcal{A} \ \vdash\ \varphi_1 \Rightarrow \varphi'_1$, ..., $\mathcal{A} \ \vdash\ \varphi_n \Rightarrow \varphi'_n$ is derivable. Thus, the rule schema contains one proof rule instance for each of the rules in C. For example, if $C = \{\varphi_1 \Rightarrow \varphi'_1,\ \varphi_2 \Rightarrow \varphi'_2\}$, the **Set circularity** proof rule schema comprises of two proof rules, one with the conclusion $\mathcal{A} \ \vdash\ \varphi_1 \Rightarrow \varphi'_1$ and the other with the conclusion $\mathcal{A} \ \vdash\ \varphi_2 \Rightarrow \varphi'_2$, both of them having the same four premises, namely:

$$\begin{array}{ll} \mathcal{A} \ \vdash \varphi_1 \Rightarrow^+ \varphi''_1 & \mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1,\ \varphi_2 \Rightarrow \varphi'_2\} \ \vdash \varphi''_1 \Rightarrow \varphi'_1 \\ \mathcal{A} \ \vdash \varphi_2 \Rightarrow^+ \varphi''_2 & \mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1,\ \varphi_2 \Rightarrow \varphi'_2\} \ \vdash \varphi''_2 \Rightarrow \varphi'_2 \end{array}$$

Therefore, **Set circularity** allows us to derive several rewrite rules at once, which is particularly useful when we want to verify mutually circular properties. It also allows us to derive *all* the desired properties at once, by simply enforcing the left-hand side of each to rewrite one step with the original language semantics and then adding all of them to the semantics for the remaining proof obligations (this is, for example, how MatchC is implemented—see Section 5). Proposition 4 states that **Set circularity** does not increase the expressiveness of our proof system, that is, every sequent that can be proved using **Set circularity** can also be proved with the original proof system in Figure 1.

Proposition 4. *Let $\mathcal{A}$ be a rewrite system and C be a finite set of rewrite rules. Then $\mathcal{A} \ \vdash\ C$ with the proof systems in Figures 1 and 9 iff $\mathcal{A} \ \vdash\ C$ with the proof system in Figure 1.*

A rigorous proof is given in Appendix A. Intuitively, this holds because one can iteratively apply the **Circularity** rule for each of the rewrite rule proof obligations in C. For example, if C consists of two rules like above, then we can first derive $\mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1\} \ \vdash\ \varphi_2 \Rightarrow \varphi'_2$ using the following instance of **Circularity**

$$\frac{\mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1\} \ \vdash \varphi_2 \Rightarrow^+ \varphi''_2 \qquad \mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1,\ \varphi_2 \Rightarrow \varphi'_2\} \ \vdash \varphi''_2 \Rightarrow \varphi'_2}{\mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1\} \ \vdash \varphi_2 \Rightarrow \varphi'_2}$$

and then use it to transform any proof derivation for

$$\mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1,\ \varphi_2 \Rightarrow \varphi'_2\} \ \vdash \varphi''_1 \Rightarrow \varphi'_1$$

into a proof derivation for $\mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1\} \ \vdash\ \varphi''_1 \Rightarrow \varphi'_1$. Then we can derive $\mathcal{A} \ \vdash\ \varphi_1 \Rightarrow \varphi'_1$ using another instance of **Circularity**,

$$\frac{\mathcal{A} \ \vdash \varphi_1 \Rightarrow^+ \varphi''_1 \qquad \mathcal{A} \cup \{\varphi_1 \Rightarrow \varphi'_1\} \ \vdash \varphi''_1 \Rightarrow \varphi'_1}{\mathcal{A} \ \vdash \varphi_1 \Rightarrow \varphi'_1}$$

and similarly for $\mathcal{A} \ \vdash\ \varphi_2 \Rightarrow \varphi'_2$.

5. Implementation and Evaluation

Here we discuss our MatchC implementation of the proof system in Figure 1. While the proof system can be easily implemented in most theorem proving environments, we preferred an implementation that emphasizes automated reasoning. Our results demonstrate that matching logic rewriting is practical in a more common sense, that is, that it can be used for relatively efficient and highly automated verification of expressive properties about challenging programs (like AVL trees and Schorr-Waite).

As discussed in Section 4, general matching logic specifications are rewrite rules between formulae. As seen in Section 2, our tool handles specifications of the form:

$$\langle \texttt{code}\ \cdots \rangle_k \wedge \pi \Rightarrow \langle\ \cdots \rangle_k \wedge \pi'$$

where π and π' are existential patterns. For now, MatchC only supports (partial correctness) rules summarizing the behavior of functions or loops. An invariant π for `while(C)S` is just syntactic sugar for a rewrite rule. For clarity, let us consider the case when the condition `C` checks if a program variable `x` is non-zero (the general case is similar). Then, if the environment of π maps `x` into $v_{\texttt{x}}$, we associate with the loop the following rule:

$$\langle \texttt{while(x)S}\ \cdots \rangle_k \wedge \pi \Rightarrow \langle\ \cdots \rangle_k \wedge \pi \wedge (v_{\texttt{x}} = 0)$$

Program	Cells	Time (s)	# paths	SMT?
Example programs				
undefined	—	0.01	1	no
list reverse	heap	0.06	2	no
list read	in, heap	0.14	7	no
stack inspection	call stack	0.24	8	no
tree to list (iterative)	heap, out	0.24	11	no
Undefined programs				
division by zero	—	0.01	1	no
uninitialized variable	—	0.01	1	no
unallocated location	—	0.01	1	no
Simple programs that need only the environment cell				
average	—	0.02	1	no
min	—	0.04	2	no
max	—	0.04	2	no
mul by add	—	0.13	3	yes
sum (recursive)	—	0.06	2	yes
sum (iterative)	—	0.08	2	yes
assoc comm	—	0.03	1	no
Lists				
list head	heap	0.02	2	no
list tail	heap	0.02	1	no
list add	heap	0.02	1	no
list swap	heap	0.03	3	no
list deallocate	heap	0.04	2	no
list length (recursive)	heap	0.05	2	no
list length (iterative)	heap	0.07	2	no
list sum (recursive)	heap	0.05	2	no
list sum (iterative)	heap	0.07	2	no
list append	heap	0.1	3	no
list copy	heap	0.13	3	no
list filter	heap	0.22	5	no
Input and output				
read write	in, out	0.12	4	no
list write	heap, out	0.06	2	no
list read write	heap, in, out	0.15	5	no
Trees				
tree height	heap	0.1	4	no
tree size	heap	0.07	3	no
tree find	heap	0.12	5	no
tree mirror	heap	0.7	3	no
tree in-order	heap	0.7	3	no
tree pre-order	heap	0.7	3	no
tree post-order	heap	0.7	3	no
tree deallocate	heap	0.14	7	no
tree to list (recursive)	heap, out	0.1	4	no
Call stack				
only g calls f	call stack	0.04	2	no
h in stack when f	call stack	0.04	2	no
Sorting algorithms				
insert	heap	0.35	5	no
insertion sort	heap	0.41	6	no
bubble sort	heap	0.30	6	no
quicksort	heap	0.47	8	no
merge sort	heap	1.97	16	yes
Search trees				
BST find	heap	0.15	5	yes
BST insert	heap	0.13	4	yes
BST delete	heap	0.38	10	yes
AVL find	heap	0.15	5	yes
AVL insert	heap	43.5	23	yes
AVL delete	heap	133.58	36	yes
Schorr-Waite				
tree Schorr Waite	heap	0.28	6	no
graph Schorr Waite	heap	1.73	8	no

Figure 10. Results of MatchC program verification

Therefore, the above rule summarizes the behavior of the loop when it terminates. Section 4.3 discusses the rewrite rule associated to the loop invariant of `reverseList` in Figure 3.

Let $\mathcal{S}$ be the rewrite system associated to the C fragment considered in this paper, and let $\mathcal{C}$ be the set of rewrite rules specifying all the user specified program properties (i.e., properties that one wants to verify). $\mathcal{C}$ contains one candidate rule for each function and one candidate rule for each loop. MatchC derives the sequent $\mathcal{S} \vdash \mathcal{C}$ using the proof system in Figures 1 and 9. It begins by applying **Set Circularity** for $\mathcal{C}$ and reduces the task to deriving individual sequents of the form $\mathcal{S} \cup \mathcal{C} \vdash \pi \Rightarrow \pi'$. To prove each such rule, the tool rewrites the formula π using rules from $\mathcal{S} \cup \mathcal{C}$ searching for a formula that implies π'. Whenever the semantics rule for **`if`** in $\mathcal{S}$ cannot apply because the condition evaluates to a symbolic value, the verifier performs case analysis and splits the formula into a disjunction of two formulae, and continues to rewrite both of them (sound according to the **Case Analysis** proof rule). Similarly, when no rule can be applied, it tries to use abstraction axioms (like the one for lists in Section 3.4), and performs case analysis to continue the rewriting. As an optimisation, when a formula can be rewritten with rules from both $\mathcal{S}$ and $\mathcal{C}$, the verifier only uses the rules from $\mathcal{C}$. In particular, only a loop without a specified invariant is unrolled, and only a function without a rule specification is invoked. Also, if the current formula implies that application of an abstraction axiom would result into a more concrete formula, the verifier applies the respective axiom (for instance, knowing the head of a linked list is not null results in an automatic list unrolling).

The semantics of the C fragment is given in $\mathbb{K}$ [44] as a set of rewrite rules $\mathcal{S}$ over the configuration in Figure 7. The deduction of rewrite rules is also implemented in $\mathbb{K}$, as a set of rewrite rules which are added to the original set of semantic rules. Checking of matching logic formulae implication (required for **Consequence**) is implemented in Maude [12]. Proving such an implication consists of two parts: matching the structure of the configuration, and checking the constraints. The structure matching is done modulo both pattern axioms and mathematical domain axioms. If all the structure is successfully matched, and the remaining constraint does not simplify to true, it is passed to CVC3 [3] and Z3 [15]. MatchC is sound w.r.t. the matching logic proof system, but is incomplete.

Figure 10 summarises the results of our experiments. Two factors guided us: proving functional correctness (as opposed to just memory safety) and doing so automatically (the user only provides the specifications). The undefined behavior is detected by execution based on the semantics. The functional behavior of the programs manipulating lists and trees and performing arithmetic and I/O operations is algebraically defined, and is similar to that of the examples in Figures 3 , 4 and 6. For the sorting algorithms, MatchC checks that the sequence is sorted and has the expected multiset of elements, and for the search trees, it checks that the tree respects the data structure invariant and has the expected multiset of elements.

The Schorr-Waite graph marking algorithm [49] computes all the nodes in a graph that are reachable from a set of starting nodes. To achieve that, it visits the graph nodes in depth-first search order, by reversing pointers on the way down, and then restoring them on the way up. Its main application is in garbage collection. Schorr-Waite algorithm presents considerable verification challenges [27, 32]. We analyzed the algorithm itself as originally given for graphs, and a simplified version in which the graph is a tree. For both cases we proved that a node is marked if and only if it is reachable from the set of initial nodes, and that the graph does not change.

Most of these examples are proved in milliseconds and do not require SMT support. We mention that the AVL insert and delete programs take approximately 3 minutes together because some of the auxiliary functions (like balance) are not given specifications and thus their bodies are being executed, resulting in a larger

number of paths to analyze. The experiments were conducted on a quad-code, 2.2GHz, 4GB machine running Linux. The source code of MatchC, as well as an online interface allowing one to verify and experiment with all C programs discussed here, or to introduce new ones, is publicly available from the matching logic web page at `http://fsl.cs.uiuc.edu/ml`.

6. Related Work

The idea of regarding a program (fragment) as a specification transformer to analyze programs in a forwards-style is very old. Floyd did it in his seminal 1967 paper [22]. However, Floyd's rules are not concerned with program structural configurations, are not meant to be operational, and introduce quantifiers.

Equational algebraic specifications have been used to express pre- and post-conditions and then verify programs forwards using term rewriting [23]. Evolving specifications [41] adapt and extend this basic idea to compositional systems, refinement and behavioral specifications. What distinguishes the various specification transforming approaches is the formalism and style used for specifications. What distinguishes matching logic is its apparently low-level formalism, which drops no detail from the program configuration. The other approaches attempt to do the opposite, to use formalisms which are as abstract as possible. Matching logic builds upon the belief that there are advantages in working with explicit configuration patterns, and that the use of variables in configurations offers a comfortable level of abstraction by only mentioning in each rule those configuration components which are necessary.

The state of the art in mechanized program verification [1, 39] is to define both an operational and an axiomatic semantics in a higher-order logic framework, where the two semantics share the definition of a "state", and to prove the axiomatic semantics sound w.r.t. the operational semantics. Then one can deduce theorems about programs using rules from both semantics. While libraries of tactics are developed to partially automate the process, it still needs to be done for each language independently. The nine-rule proof system proposed in this paper is language-independent and it should be easy to mechanize in a higher-order framework.

Separation logic [40, 42] is an extension of Hoare logic. There is a major difference between separation and matching logic rewriting: the former enhances Hoare logic to work better with heaps, while the latter provides an alternative to Hoare logics in which the configuration structure is explicit in the specifications, so heaps are treated uniformly just like any other structures in the configuration.

Bedrock [11] is a framework which uses computational higher-order separation logic and supports mostly-automated proofs about low-level programs. Unlike MatchC, Bedrock requires the user to annotate the source code with hints for lemma applications (like list rolling and unrolling). Specifications use operators defined in a pure functional language, similarly to the operators defined algebraically in matching logic. It is likely that the tactics employed by Bedrock could be adapted for higher-order matching logic.

Shape analysis [48] allows one to examine and verify properties of heap structures. It has been shown to be quite powerful when reasoning about heaps. The ideas of shape analysis have also been combined with those of separation logic [16] to quickly infer invariants for programs operating on lists. They can likely be also combined with matching logic in order to infer patterns.

Dynamic logic (DL) [24] extends FOL with modal operators to embed program fragments within program specifications. For example, the formula $\varphi \rightarrow [\mathtt{s}]\varphi'$ has the meaning "after executing $\mathtt{s}$ in a state satisfying φ, a state may be reached which satisfies φ'". Like in matching logic, programs and specifications also coexits in the same logic in dynamic logic. However, matching logic achieves that by staying within FOL and making use of FOL's algebraic signatures and term models. The dynamic logic based KeY project [4] has many common goals and similarities with matching logic; specifically they both attempt to serve as alternatives, rather than extend, conventional axiomatic semantics, and their current implementations are both based on dynamic language semantics.

There are many Hoare-logic-based verification frameworks, such as ESC/Java [21], VCC [13], Spec# [2], HAVOC [30] and Dafny [31]. Frama-C/Why [20, 27] proved many properties relating to the Schorr-Waite algorithm. However, their proofs were not entirely automated. The weakness of traditional Hoare-like approaches is that reasoning about non-inductively defined data-types and about heap structures tend to be difficult, requiring extensive manual intervention in the proof process. Jahob [51] is another verification framework that mixes automated and interactive reasoning. Among the separation-logic-based tools, we mention SLAyer [6], Xisa [10] and Thor [34], which automatically check memory safety, shape and/or arithmetic properties, and Smallfoot [5], Hip [38] and Verifast [28], which can prove functional correctness.

7. Conclusion and Future Work

Matching logic rewriting is a semantic framework based on rewrite rules between matching logic formulae. It smoothly integrates operational and axiomatic semantics, allowing designers to define programming language semantics operationally, using state of the art term rewriting techniques and tools, and then to turn these formal executable semantics into program verification logics. A proof system was introduced and shown sound and complete for operational semantics, and partially correct for program verification. A prototype verifier for a fragment of C has been implemented and evaluated with encouraging results.

We have only scratched the surface, much is left to be done. First, matching logic itself is new, so we need to develop powerful reasoning engines for it. Concurrency and non-determinism were purposely left out; these are big topics which deserve full attention; rewriting logic was designed specifically to deal with concurrency and non-determinism, so we are optimistic, though. Relative completeness and total correctness also need to be addressed. Matching logic makes formal language semantics useful for verification, so we expect that many real languages will be given rewriting semantics. Some of these languages have already been given complete semantics, others were just started. Like other formal semantics, matching logic can also be embedded into higher-level formalisms and theorem provers, so that proofs of relationships to other semantics can be mechanized, or even programs verified.

References

[1] A. W. Appel. Verified software toolchain. In *ESOP*, pages 1–17, 2011.

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[3] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

[4] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[6] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[7] J. A. Bergstra and J. V. Tucker. Initial and final algebra semantics for data type specifications: Two characterization theorems. *SIAM J. Comput.*, 12(2):366–387, 1983.

[8] G. Berry and G. Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[9] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009.

[10] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[11] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.

[12] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. 2007.

[13] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, 2009.

[14] O. Danvy and L. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, 2004.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[16] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.

[17] C. Ellison and G. Roşu. A formal semantics of C with applications. In *POPL*, 2012. To appear.

[18] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Th. Comp. Sci.*, 103(2):235–271, 1992.

[19] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN 978-0-262-06275-6.

[20] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[22] R. W. Floyd. Assigning meaning to programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[23] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[24] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, 2008.

[27] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *SEFM*, pages 190–199, 2005.

[28] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55, 2011.

[29] C. Klein, J. McCarthy, S. Jaconette, and R. B. Findler. A semantics for context-sensitive reduction semantics. In *APLAS*, 2011. To appear.

[30] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.

[31] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.

[32] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS*, 2006.

[33] D. B. MacQueen and D. Sannella. Completeness of proof systems for equational specifications. *IEEE Trans. Software Eng.*, 11(5):454–461, 1985.

[34] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.

[35] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[36] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.

[37] P. D. Mosses. *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004.

[38] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[39] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[40] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symb. Logic*, 5(2):215–244, 1999.

[41] D. Pavlovic and D. R. Smith. Composition and refinement of behavioral specifications. In *ASE*, pages 157–165, 2001.

[42] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[43] G. Rosu and D. Lucanu. Circular coinduction: A proof theoretical foundation. In *CALCO*, pages 127–144, 2009.

[44] G. Rosu and T.-F. Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[45] G. Rosu and A. Stefanescu. Matching logic: a new program verification approach (NIER track). In *ICSE*, pages 868–871, 2011.

[46] G. Rosu, W. Schulte, and T.-F. Serbanuta. Runtime verification of C memory safety. In *RV*, pages 132–151, 2009.

[47] G. Rosu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, pages 142–162, 2010.

[48] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3):217–298, 2002.

[49] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[50] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential java programs. In *WADT*, pages 1–21, 1999.

[51] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351, 2009.

A. Proofs

A.1 Proof Proposition 1

Proposition 1. *If φ is a matching logic formula, $\rho : Var \to \mathcal{T}$ and $\gamma \in \mathcal{T}_{Cfg}$, then $(\gamma, \rho) \models \varphi$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi^\Box$. Also, $\models \varphi$ iff $\mathcal{T} \models_{\text{FOL}_=} \varphi^\Box$.*

Proof: We prove the first part of Proposition 1 by induction on the structure of the formula φ. We distinguish the following cases:

- φ is $t = t'$. Then $(\gamma, \rho) \models \varphi$ iff $\rho(t) = \rho(t')$ iff $\rho^\gamma(t) = \rho^\gamma(t')$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi^\Box$ (we use the fact that t and t' are terms and thus do not contain $\Box$).
- φ is $\forall X \varphi'$. By the induction hypothesis, for all substitutions ρ' that agree with ρ on $Var \setminus X$, we have that $(\gamma, \rho') \models \varphi'$ iff $\rho'^\gamma \models_{\text{FOL}_=} \varphi'^\Box$, hence $(\gamma, \rho) \models \varphi$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi^\Box$, and we are done.
- φ is $\varphi_1 \wedge \varphi_2$. By the induction hypothesis, $(\gamma, \rho) \models \varphi_1$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi_1^\Box$ and $(\gamma, \rho) \models \varphi_2$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi_2^\Box$. Then, $(\gamma, \rho) \models \varphi$ iff $(\gamma, \rho) \models \varphi_1$ and $(\gamma, \rho) \models \varphi_2$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi_1^\Box$ and $\rho^\gamma \models_{\text{FOL}_=} \varphi_2^\Box$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi^\Box$, and we are done.
- φ is $\neg \varphi'$. By the induction hypothesis, $(\gamma, \rho) \models \varphi'$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi'^\Box$. Then we have that $(\gamma, \rho) \models \varphi$ iff $(\gamma, \rho) \not\models \varphi'$ iff $\rho^\gamma \not\models_{\text{FOL}_=} \varphi'^\Box$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi^\Box$, and we are done.
- φ is π. Then $(\gamma, \rho) \models \varphi$ iff $\gamma = \rho(\pi)$ iff $\rho^\gamma(\Box) = \rho^\gamma(\pi)$ iff $\rho^\gamma \models_{\text{FOL}_=} \Box = \pi$, and we are done.

For the second part of Proposition 1, we notice that proving $\models \varphi$ iff $\mathcal{T} \models_{\text{FOL}_=} \varphi^\Box$ is equivalent to proving that, for all γ and ρ, it is the case that $(\gamma, \rho) \models \varphi$ iff $\rho^\gamma \models_{\text{FOL}_=} \varphi^\Box$, which follows from the first part of Proposition 1, and we are done. $\Box$

A.2 Proof Proposition 2

Proposition 2. ***(operational soundness)*** *If $\mathcal{S}$ is well-defined, $\mathcal{S} \vdash \varphi_{LHS} \Rightarrow^{1\text{-}8} \varphi_{RHS}$, and $(\gamma, \rho) \models \varphi_{LHS}$ for some $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$, then there is some $\gamma' \in \mathcal{T}_{Cfg}$ with $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_{RHS}$.*

Proof: Let $\mathcal{P}$ be a matching logic proof tree deriving the sequent $\mathcal{S} \vdash \varphi_{LHS} \Rightarrow \varphi_{RHS}$. We prove Proposition 2 by induction on the structure of $\mathcal{P}$. Let up pick $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi_{LHS}$. We distinguish the following cases:

- $\mathcal{P}$ is a **Reflexivity** step:

$$\frac{}{\mathcal{S} \vdash \varphi \Rightarrow \varphi}$$

  We pick γ' to be γ. Trivially, we have that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$, and $(\gamma', \rho) \models \varphi$, so we are done.
- $\mathcal{P}$ is an **Axiom** step:

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{S}}{\mathcal{S} \vdash \varphi \Rightarrow \varphi'}$$

  Since $\mathcal{S}$ is well-defined, there exists a γ' such that $(\gamma', \rho) \models \varphi'$. By the definition of the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, we have that $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$, which implies that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$, and we are done.
- The last step in $\mathcal{P}$ is a **Substitution** step:

$$\frac{\mathcal{S} \vdash \varphi \Rightarrow \varphi' \quad \theta : Var \to \mathcal{T}_\Sigma(Var)}{\mathcal{S} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

  Let $\rho' = \theta(\rho)$. It follows that $(\gamma, \rho') \models \varphi$. By the induction hypothesis there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho') \models \varphi'$. Then, we have that $(\gamma', \rho) \models \theta(\varphi')$, and we are done.
- The last step in $\mathcal{P}$ is a **Transitivity** step:

$$\frac{\mathcal{S} \vdash \varphi_1 \Rightarrow \varphi_2 \quad \mathcal{S} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{S} \vdash \varphi_1 \Rightarrow \varphi_3}$$

  By the induction hypothesis, there exists some γ'' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma''$ and $(\gamma'', \rho) \models \varphi_2$. Also by the induction hypothesis, there exists a γ' such that $\gamma'' \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_3$. Due to the transitivity of the transition relation $\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$ in $\mathcal{T}$, it follows that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$, and we are done.
- The last step in $\mathcal{P}$ is a **Case analysis** step:

$$\frac{\mathcal{S} \vdash \varphi_1 \Rightarrow \varphi \quad \mathcal{S} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{S} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

  By the definition of satisfaction for disjunction, $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$ implies $(\gamma, \rho) \models \varphi_1$ or $(\gamma, \rho) \models \varphi_2$. We can assume without loss of generality that $(\gamma, \rho) \models \varphi_1$. By the induction hypothesis there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi$, and we are done.
- The last step in $\mathcal{P}$ is a **Logic framing** step:

$$\frac{\mathcal{S} \vdash \varphi \Rightarrow \varphi' \quad \psi \text{ is a FOL}_= \text{ formula}}{\mathcal{S} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

  By the definition of satisfaction for conjunction and FOL formulae, $(\gamma, \rho) \models \varphi \wedge \psi$ implies that $(\gamma, \rho) \models \varphi$ and $\rho \models \psi$. By the induction hypothesis, there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. It follows that $(\gamma', \rho) \models \varphi' \wedge \psi$, and we are done.
- The last step in $\mathcal{P}$ is a **Consequence** step:

$$\frac{\models \varphi_1 \to \varphi_1' \quad \mathcal{S} \vdash \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \to \varphi_2}{\mathcal{S} \vdash \varphi_1 \Rightarrow \varphi_2}$$

  Since $\models \varphi_1 \to \varphi_1'$ and $(\gamma, \rho) \models \varphi_1$, it follows that $(\gamma, \rho) \models \varphi_1'$. By the induction hypothesis, there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_2'$. Since $\models \varphi_2' \to \varphi_2$, it follows that $(\gamma', \rho) \models \varphi_2$, and we are done.
- the last step in $\mathcal{P}$ is an **Abstraction** step:

$$\frac{\mathcal{S} \vdash \varphi \Rightarrow \varphi' \quad X \cap FreeVars(\varphi') = \emptyset}{\mathcal{S} \vdash \exists X\, \varphi \Rightarrow \varphi'}$$

  Since the variables in X do not appear free in φ', and $(\gamma, \rho) \models \exists X \varphi$, by the definition of the satisfaction for existential quantification, there exists a substitution ρ' that agrees with ρ on the free variables of φ', such that $(\gamma, \rho') \models \varphi$. By the induction hypothesis, there exists γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho') \models \varphi'$. Since ρ' and ρ agree on the free variables of φ', we can conclude that $(\gamma', \rho) \models \varphi'$, and we are done.

$\Box$

A.3 Proof Proposition 3

Proposition 3. ***(operational completeness)*** *If all the rewrite rules in $\mathcal{S}$ only use conjunctive patterns, $\mathcal{T}$ is chosen to be a term model, and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ for some $\gamma, \gamma' \in \mathcal{T}_{Cfg}$, then $\mathcal{S} \vdash \gamma \Rightarrow^{1\text{-}4} \gamma'$.*

Proof: We prove Proposition 3 by induction on the length n of the rewrite sequence $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$. We mention that in doing so we only need to use the first four matching logic proof rules.
Base Case $n = 0$. It follows that $\gamma = \gamma'$. By applying the **Reflexivity** proof rule for γ, it follows that $\mathcal{S} \vdash \gamma \Rightarrow \gamma$.
Induction Step $n > 0$. We choose γ'' such that $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma''$, and $\gamma'' \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$. By the induction hypothesis we have that $\mathcal{S} \vdash \gamma'' \Rightarrow \gamma'$. By the definition of the $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ transition system, it follows that there exists a rule $\varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$, and $(\gamma', \rho) \models \varphi'$. Since $\mathcal{T}$ is a term model, and due to the fact that rule $\varphi \Rightarrow \varphi'$ has only conjunctive patterns, we can pick a (ground) substitution $\theta : Var \to \mathcal{T}_\Sigma$ such that $\theta(\varphi) = \gamma$ and $\theta(\varphi') = \gamma''$. By using the **Axiom** and **Substitution** proof rules, we can derive that $\mathcal{S} \vdash \gamma \Rightarrow \gamma''$. Finally, by **Transitivity**, we can derive that $\mathcal{S} \vdash \gamma \Rightarrow \gamma'$, and we are done. $\Box$

A.4 Proof Theorem 2

THEOREM 1. ***(partial correctness)*** *Let $\mathcal{S}$ be a well-defined and deterministic set of rewrite rules, and $\mathcal{S} \vdash \varphi_{LHS} \Rightarrow \varphi_{RHS}$ a sequent derived with the proof system in Figure 1. Then $\mathcal{S} \models \varphi_{LHS} \Rightarrow \varphi_{RHS}$.*

Proof: Recall that $\mathcal{T}$ is a fixed configuration model, and $\mathcal{S}$ a fixed finite, well-defined and deterministic set of rewrite rules. We define the relation $(\mathcal{T}, <^{\mathcal{S}})$ as follows:

$$\gamma_1 <^{\mathcal{S}} \gamma_2 \text{ iff } \gamma_1 \text{ and } \gamma_2 \text{ terminate in } (\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}}) \text{ and } \gamma_2 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$$

We notice that $<^{\mathcal{S}}$ has the following properties:

irreflexivity if $\gamma \in \mathcal{T}$ terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, then $\neg(\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma)$, hence $\neg(\gamma <^{\mathcal{S}} \gamma)$;

asymmetry if $\gamma_1, \gamma_2 \in \mathcal{T}$ terminate in $(\mathcal{T} \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, and $\gamma_1 <^{\mathcal{S}} \gamma_2$, then $\gamma_2 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$ and $\neg(\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2)$, hence $\neg(\gamma_2 <^{\mathcal{S}} \gamma_1)$;

transitivity if $\gamma_1, \gamma_2, \gamma_3 \in \mathcal{T}$ terminate in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, and $\gamma_1 <^{\mathcal{S}} \gamma_2$ and $\gamma_2 <^{\mathcal{S}} \gamma_3$, then $\gamma_2 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$ and $\gamma_3 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$; by the transitivity of $\Rightarrow_{\mathcal{S}}^{+\mathcal{T}}$, it follows that $\gamma_3 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_1$, hence $\gamma_1 <^{\mathcal{S}} \gamma_3$.

We can conclude that $<^{\mathcal{S}}$ is a partial order relation. Moreover, since any decreasing chain has associated a rewrite sequence containing only terms which terminate in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, it follows that there are no infinite decreasing chains, or equivalently, that $<^{\mathcal{S}}$ is well-founded.

We consider the domain $\mathcal{D}$ of triples

$$(\gamma, \mathcal{A}, \mathcal{P})$$

where $\gamma \in \mathcal{T}$ is a ground configuration that terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, $\mathcal{A}$ is a finite set of rewrite rules, and $\mathcal{P}$ is a matching logic proof tree deriving a sequent of the form $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$. We define the lexicographical order $(\mathcal{D}, \prec)$ as follows:

$$\begin{array}{ll} & (\gamma_1, \mathcal{A}_1, \mathcal{P}_1) \prec (\gamma_2, \mathcal{A}_2, \mathcal{P}_2) \\ \text{iff} & \gamma_1 <^{\mathcal{S}} \gamma_2 \\ \text{or} & \gamma_1 = \gamma_2 \text{ and } \mathcal{A}_1 \subsetneq \mathcal{A}_2 \\ \text{or} & \gamma_1 = \gamma_2 \text{ and } \mathcal{A}_1 = \mathcal{A}_2 \text{ and } \mathcal{P}_1 \text{ is a proper subtree of } \mathcal{P}_2 \end{array}$$

We notice that $\prec$ is a lexicographic order based on three well-founded partial order relations, namely $(\mathcal{T}, <^{\mathcal{S}})$, strict inclusion of finite sets of rules ($\subsetneq$) and proper subtree relation on proof trees. It is known that the lexicographic ordering of sequences of fixed length [2] based on well-founded orders for each component of the sequence is itself well-founded, thus $\prec$ is well-founded.

DEFINITION 9. *If $(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D}$ is such that $\mathcal{P}$ derives the sequent $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$, then let $Prop(\gamma, \mathcal{A}, \mathcal{P})$ be the following property: for all $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$, there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi$; moreover, if $\mathcal{P}$ does not use the* **Reflexivity** *rule, or the last step in $\mathcal{P}$ is a* **Circularity** *step, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ (that is, γ rewrites to γ' in one or more steps).*

Let $\mathcal{P}_{fixed}$ be a fixed matching logic proof tree deriving the sequent $\mathcal{S} \vdash \varphi_{LHS} \Rightarrow \varphi_{RHS}$. We define $\mathcal{D}_{fixed}$ as

$$\mathcal{D}_{fixed} = \{(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D} \mid \mathcal{P} \text{ is a subtree of } \mathcal{P}_{fixed}\}$$

With these definitions, it is easy to see that Theorem 2 becomes a corollary of the following result: for all $\gamma \in \mathcal{T}$ such that γ terminates in $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, the property $Prop(\gamma, \mathcal{S}, \mathcal{P}_{fixed})$ holds. Also, note that the triple $(\gamma, \mathcal{S}, \mathcal{P}_{fixed})$ belongs to $\mathcal{D}_{fixed}$ for all the terminating γ. These observations imply that it suffices to prove the following more general result:

[2] Note that this is not true for sequences or arbitrary lengths. For example, the set of finite sequences of elements of $(\{0, 1\}, <)$ is not well-founded, even if $<$ is well-founded on $\{0, 1\}$.

LEMMA 1. *$Prop(\gamma, \mathcal{A}, \mathcal{P})$ holds for all $(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D}_{fixed}$.*

We prove Lemma 1 by well-founded induction on the $\prec$ partial order. Let us pick a $(\gamma, \mathcal{A}, \mathcal{P}) \in \mathcal{D}_{fixed}$ and let $\rho : Var \to \mathcal{T}$ be such that (γ, ρ) satisfy the left-hand-side formula of the rule derived by $\mathcal{P}$. We have the following cases based on the structure of $\mathcal{P}$:

- $\mathcal{P}$ is a **Reflexivity** step:

$$\frac{}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

  We pick γ' to be γ. Trivially, we have that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$, and $(\gamma', \rho) \models \varphi$, so we are done.

- $\mathcal{P}$ is an **Axiom** step:

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

  We distinguish two cases:

  - $\varphi \Rightarrow \varphi'$ belongs to $\mathcal{S}$. Since $\mathcal{S}$ is well-defined, there exists a γ' such that $(\gamma', \rho) \models \varphi'$. By the definition of the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, we have that $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$, which implies that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$, and we are done.
  - $\varphi \Rightarrow \varphi'$ belongs to $\mathcal{A} \setminus \mathcal{S}$. Since the only way to add rules to the set of axioms is by using the **Circularity** rule, there must be a set of axioms $\mathcal{A}'$ and a proof tree $\mathcal{P}'$ deriving the sequent $\mathcal{A}' \vdash \varphi \Rightarrow \varphi'$, such that $\mathcal{P}'$ is a subtree of $\mathcal{P}_{fixed}$, and $\mathcal{P}$ is a leaf of $\mathcal{P}'$. Rules cannot be dropped from the set of axioms as we traverse a matching logic proof tree bottom-up, hence it must be the case that $\mathcal{A}' \subsetneq \mathcal{A}$. It follows that $(\gamma, \mathcal{A}', \mathcal{P}') \prec (\gamma, \mathcal{A}, \mathcal{P})$. By the induction hypothesis, $Prop(\gamma, \mathcal{A}', \mathcal{P}')$ holds. The last step in $\mathcal{P}'$ is a **Circularity** step, hence there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$, so we are done.

- The last step in $\mathcal{P}$ is a **Substitution** step:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad \theta : Var \to \mathcal{T}_{\Sigma}(Var)}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

  Let $\rho' = \theta(\rho)$. It follows that $(\gamma, \rho') \models \varphi$. By the induction hypothesis there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho') \models \varphi'$. Then, we have that $(\gamma', \rho) \models \theta(\varphi')$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

- The last step in $\mathcal{P}$ is a **Transitivity** step:

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}$$

  Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be the proof trees deriving $\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2$ and $\mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3$. Notice that $\mathcal{P}_1$ is a subtree of $\mathcal{P}$, hence $(\gamma, \mathcal{A}, \mathcal{P}_1) \prec (\gamma, \mathcal{A}, \mathcal{P})$. By the induction hypothesis, there exists a γ'' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma''$ and $(\gamma'', \rho) \models \varphi_2$. Since $\mathcal{P}_2$ is also a subtree of $\mathcal{P}$ and either $\gamma'' <^{\mathcal{S}} \gamma$ or $\gamma'' = \gamma$, it follows that $(\gamma'', \mathcal{A}, \mathcal{P}_2) \prec (\gamma, \mathcal{A}, \mathcal{P})$. Also by the induction hypothesis, there exists a γ' such that $\gamma'' \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_3$. Due to the transitivity of the transition relation $\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$ in $\mathcal{T}$, it follows that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma''$ and $\gamma'' \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ by the induction hypothesis. This implies that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$, and we are done.

- The last step in $\mathcal{P}$ is a **Case analysis** step:

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

  By the definition of satisfaction for disjunction, $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$ implies $(\gamma, \rho) \models \varphi_1$ or $(\gamma, \rho) \models \varphi_2$. We can assume without loss of generality that $(\gamma, \rho) \models \varphi_1$. By the induction hypothesis there

exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

- The last step in $\mathcal{P}$ is a **Logic framing** step:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad \psi \text{ is a FOL}_= \text{ formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

By the definition of satisfaction for conjunction and FOL formulae, $(\gamma, \rho) \models \varphi \wedge \psi$ implies that $(\gamma, \rho) \models \varphi$ and $\rho \models \psi$. By the induction hypothesis, there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. It follows that $(\gamma', \rho) \models \varphi' \wedge \psi$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

- The last step in $\mathcal{P}$ is a **Consequence** step:

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Since $\models \varphi_1 \rightarrow \varphi_1'$ and $(\gamma, \rho) \models \varphi_1$, it follows that $(\gamma, \rho) \models \varphi_1'$. By the induction hypothesis, there exists a γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi_2'$. Since $\models \varphi_2' \rightarrow \varphi_2$, it follows that $(\gamma', \rho) \models \varphi_2$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$by the induction hypothesis, and we are done.

- the last step in $\mathcal{P}$ is an **Abstraction** step:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash \exists X\, \varphi \Rightarrow \varphi'}$$

Since the variables in X do not appear free in φ', and $(\gamma, \rho) \models \exists X \varphi$, by the definition of the satisfaction for existential quantification, there exists a substitution ρ' that agrees with ρ on the free variables of φ', such that $(\gamma, \rho') \models \varphi$. By the induction hypothesis, there exists γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho') \models \varphi'$. Since ρ' and ρ agree on the free variables of φ', we can conclude that $(\gamma', \rho) \models \varphi'$. We notice that if $\mathcal{P}$ does not use the **Reflexivity** rule, then $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ by the induction hypothesis, and we are done.

- the last step in $\mathcal{P}$ is a **Circularity** step:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'' \quad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Let $\mathcal{P}^+$ and $\mathcal{P}'$ be the proof trees for the sequents $\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi''$ and $\mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'$. Notice that $\mathcal{P}^+$ is a subtree of $\mathcal{P}$, hence $(\gamma, \mathcal{A}, \mathcal{P}^+) \prec (\gamma, \mathcal{A}, \mathcal{P})$. By the induction hypothesis, $\mathit{Prop}(\gamma, \mathcal{A}, \mathcal{P}^+)$ holds. Since $\mathcal{P}^+$ does not use the **Reflexivity** rule, there exists a γ'' such that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma''$ and $(\gamma'', \rho) \models \varphi''$. It follows that $\gamma'' <^{\mathcal{S}} \gamma$, so $(\gamma'', \mathcal{A} \cup \{\varphi'' \Rightarrow \varphi'\}, \mathcal{P}') \prec (\gamma, \mathcal{A}, \mathcal{P})$. Then $\mathit{Prop}(\gamma'', \mathcal{A} \cup \{\varphi'' \Rightarrow \varphi'\}, \mathcal{P}')$ holds by the induction hypothesis. Hence, there exists γ' such that $\gamma'' \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. Due to the transitivity of the transition relation $\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$ in $\mathcal{T}$, it follows that $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma''$, and we are done.

□

A.5 Proof Prop 4

Proposition 4. *Let $\mathcal{A}$ be a rewrite system and C be a finite set of rewrite rules. Then $\mathcal{A} \vdash C$ with the proof systems in Figures 1 and 9 iff $\mathcal{A} \vdash C$ with the proof system in Figure 1.*

Proof: For brevity, we use $\mathcal{A} \vdash C$ to mean that the combined proof system in Figures 1 and 9 derives each rule in C, and use $\mathcal{A} \vdash^{\circ} C$ to mean that the proof system in Figure 1 derives each rule in C. We use $\mathcal{A} \vdash^{\circ} \varphi \Rightarrow \varphi'$ and $\mathcal{A} \vdash^{\circ} \varphi \Rightarrow^{+} \varphi'$ in a similar way.

With the above notation, the direct implication becomes $\mathcal{A} \vdash^{\circ} C$ if $\mathcal{A} \vdash C$. The proof proceeds by induction on the structure of the proof forest deriving the sequents in $\mathcal{A} \vdash C$ (one proof tree per sequent). We can assume without loss of generality that C can be partitioned into rules ($\uplus$ stands for disjoint union):

$$C = \biguplus_i \{\varphi_i \Rightarrow \varphi_i'\} \uplus \biguplus_j \{\varphi_j \Rightarrow \varphi_j'\}$$

such that each $\mathcal{A} \vdash \varphi_i \Rightarrow \varphi_i'$ and each $\mathcal{A} \vdash \varphi_j \Rightarrow \varphi_j'$ is derivable, the last step of each $\mathcal{A} \vdash \varphi_i \Rightarrow \varphi_i'$ is a step in Figure 1, and the last step of each $\mathcal{A} \vdash \varphi_j \Rightarrow \varphi_j'$ is a **Set circularity** step. By the induction hypothesis, all the prerequisites of each $\mathcal{A} \vdash \varphi_i \Rightarrow \varphi_i'$ are derivable with the proof system in Figure 1, hence each $\mathcal{A} \vdash^{\circ} \varphi_i \Rightarrow \varphi_i'$ is derivable. Also by the induction hypothesis, all the prerequisites of each $\mathcal{A} \vdash \varphi_j \Rightarrow \varphi_j'$ are derivable with the proof system in Figure 1. It follows from Lemma 3 (see below) that each $\mathcal{A} \vdash^{\circ} \varphi_j \Rightarrow \varphi_j'$ is derivable, which means $\mathcal{A} \vdash^{\circ} C$. The converse implication is trivial, and we are done.

We therefore have to prove Lemma 3. For it, we need the following natural deduction property:

Lemma 2. *If $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ and $\mathcal{A} \subseteq \mathcal{A}'$ then $\mathcal{A}' \vdash \varphi \Rightarrow \varphi'$*

Lemma 2 follows trivially by induction on the structure of the proof tree deriving the sequent.

Lemma 3. *Let $\mathcal{A}$ be a rewrite system and C be a finite set of rules $\{\varphi_1 \Rightarrow \varphi_1', \ldots, \varphi_n \Rightarrow \varphi_n'\}$ such that each sequent in $\mathcal{A} \vdash^{\circ} \{\varphi_1 \Rightarrow^{+} \varphi_1'', \ldots, \varphi_n \Rightarrow^{+} \varphi_n''\}$ and $\mathcal{A} \cup C \vdash^{\circ} \{\varphi_1'' \Rightarrow \varphi_1', ..., \varphi_n'' \Rightarrow \varphi_n'\}$ is derivable. Then $\mathcal{A} \cup (C \setminus C') \vdash^{\circ} C'$ for any $C' \subseteq C$.*

In the proof of Proposition 4 we need the particular case when $C' = C$ in order to prove that $\mathcal{A} \vdash^{\circ} \varphi_j \Rightarrow \varphi_j'$ is derivable, where $\varphi_j \Rightarrow \varphi_j' \in C$. The proof of Lemma 3 proceeds by induction on $|C'|$.
Base Case $|C'| = 0$. Then there does not exist any $\varphi_i \Rightarrow \varphi_i' \in C'$, hence each sequent in $\mathcal{A} \cup (C \setminus C') \vdash^{\circ} C'$ is trivially derivable.
Induction Step $|C'| \neq 0$. Then there exists a rule $\varphi_i \Rightarrow \varphi_i' \in C'$. Let $C'' = (C \setminus C') \cup \{\varphi_i \Rightarrow \varphi_i'\}$. Let $\overline{\mathcal{P}}$ be a (possibly partial) proof tree deriving the sequent $\mathcal{A} \cup C'' \vdash^{\circ} \varphi_i'' \Rightarrow \varphi_i'$ obtained from a proof tree deriving the sequent $\mathcal{A} \cup C \vdash^{\circ} \varphi_i'' \Rightarrow \varphi_i'$. Note that $\overline{\mathcal{P}}$ can be partial because some of the proof tasks of the form $\mathcal{A}' \vdash^{\circ} \varphi \Rightarrow \varphi'$ previously discharged by using the **Axiom** rule are now pending if $\varphi \Rightarrow \varphi' \notin \mathcal{A}'$. Rules cannot be dropped from the set of axioms as we traverse a tree bottom-up, hence $\mathcal{A} \cup C'' \subseteq \mathcal{A}'$. It follows that $\varphi \Rightarrow \varphi' \notin \mathcal{A}'$ only if $\varphi \Rightarrow \varphi' \in C \setminus C''$, that is, only if $\varphi \Rightarrow \varphi' \in C' \setminus \{\varphi_i \Rightarrow \varphi_i'\}$. By the induction hypothesis, we have that $\mathcal{A} \cup C'' \vdash^{\circ} \varphi \Rightarrow \varphi'$. By Lemma 2, it follows that $\mathcal{A}' \vdash^{\circ} \varphi \Rightarrow \varphi'$, and we can discharge all the pending proof tasks of $\overline{\mathcal{P}}$. Hence $\mathcal{A} \cup C'' \vdash^{\circ} \varphi_i'' \Rightarrow \varphi_i'$. Since $\mathcal{A} \vdash^{\circ} \varphi_i \Rightarrow^{+} \varphi_i''$, by Lemma 2, it follows that $\mathcal{A} \cup (C \setminus C') \vdash^{\circ} \varphi_i \Rightarrow^{+} \varphi_i''$. We apply the **Circularity** proof rule to derive $\mathcal{A} \cup (C \setminus C') \vdash^{\circ} \varphi_i \Rightarrow \varphi_i'$. Since $\varphi_i \Rightarrow \varphi_i' \in C'$ is arbitrarily chosen, it follows that each sequent in $\mathcal{A} \cup (C \setminus C') \vdash^{\circ} C'$ is derivable, and we are done. □

B. Verification of the `while` loop in function `reverseList` in Figure 3

Figure 11 gives a detailed formal proof of correctness for the **`while`** loop in Figure 3. The left column introduces notation for a clear and succinct proof. We use φ for a matching logic formulae, and ψ for a $\text{FOL}_=$ formulae. The variables k, ρ, σ and c stand for the frames of the k, env, heap and cfg cells. The right column contains the actual proof. Each proof line consists of the derived sequent, the proof rule applied and the prerequisites (if any). We use the acronym **ASLF** to for the successive application of **Axiom**, **Substitution** and **Logic framing**. Such a sequence is typical when taking one execution step according to the semantics of the C fragment. Notice that step 10 of the proof uses the list axiom in Section 3.4 from left-to-right, while step 20 uses the same axiom from right-to-left. The behaviour specification of the loop is captured by the rule $\exists X_1 \varphi_1 \Rightarrow \exists X_1 \varphi_{19}$.

$while_1 \equiv$ `while(x != NULL) { y = p->next; x->next = p; p = x; x = y; }`
$stmt_1 \equiv$ `x = y;` $while_1$
$stmt_2 \equiv$ `p = x;` $stmt_1$
$stmt_3 \equiv$ `x->next = p;` $stmt_2$
$stmt_4 \equiv$ `y = x->next;` $stmt_3$
$env_1 \equiv$ `p` $\mapsto$?p, `x` $\mapsto$?x, `y` $\mapsto$?y, ρ
$heap_1 \equiv$ list(?p)(?B), list(?x)(?C), σ
$\psi_1 \equiv$ A = rev(?B)@?C
$\varphi_1 \equiv \langle\langle while_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_1\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_1$
$if_1 \equiv$ `if(x != NULL)` $stmt_4$
$\varphi_2 \equiv \langle\langle if_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_1\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_1$
$if_2 \equiv$ `if(?x != NULL)` $stmt_4$
$\varphi_3 \equiv \langle\langle if_2 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_1\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_1$
$\varphi_4 \equiv \varphi_3 \wedge ?x \neq 0$
$\varphi_5 \equiv \varphi_3 \wedge ?x = 0$
$if_3 \equiv$ `if(1)` $stmt_4$
$\varphi_6 \equiv \langle\langle if_3 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_1\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_1 \wedge ?x \neq 0$
$if_4 \equiv$ `if(0)` $stmt_4$
$\varphi_7 \equiv \langle\langle if_4 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_1\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_1 \wedge ?x = 0$
$heap_2 \equiv$ list(?p)(?B), ?x $\mapsto$ [?v, ?n], list(?n)(?D), σ
$\psi_2 \equiv \psi_1 \wedge ?x \neq 0 \wedge ?C = [?v]@?D$
$\varphi_8 \equiv \langle\langle if_3 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_2\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$\varphi_9 \equiv \langle\langle stmt_4 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_2\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$\varphi_{10} \equiv \langle\langle$ `y = ?n;` $stmt_3 \curvearrowright k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_2\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$env_2 \equiv$ `p` $\mapsto$?p, `x` $\mapsto$?x, `y` $\mapsto$?n, ρ
$\varphi_{11} \equiv \langle\langle stmt_3 \curvearrowright k\rangle_{\mathsf{k}}\langle env_2\rangle_{\mathsf{env}}\langle heap_2\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$\varphi_{12} \equiv \langle\langle$ `x->next = ?p;` $stmt_2 \curvearrowright k\rangle_{\mathsf{k}}\langle env_2\rangle_{\mathsf{env}}\langle heap_2\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$heap_3 \equiv$ list(?p)(?B), ?x $\mapsto$ [?v, ?p], list(?n)(?D), σ
$\varphi_{13} \equiv \langle\langle stmt_2 \curvearrowright k\rangle_{\mathsf{k}}\langle env_2\rangle_{\mathsf{env}}\langle heap_3\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$\varphi_{14} \equiv \langle\langle$ `p = ?x;` $stmt_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_2\rangle_{\mathsf{env}}\langle heap_3\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$env_3 \equiv$ `p` $\mapsto$?x, `x` $\mapsto$?x, `y` $\mapsto$?n, ρ
$\varphi_{15} \equiv \langle\langle stmt_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_3\rangle_{\mathsf{env}}\langle heap_3\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$\varphi_{16} \equiv \langle\langle$ `x = ?n;` $while_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_3\rangle_{\mathsf{env}}\langle heap_3\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$env_4 \equiv$ `p` $\mapsto$?x, `x` $\mapsto$?n, `y` $\mapsto$?n, ρ
$\varphi_{17} \equiv \langle\langle while_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_4\rangle_{\mathsf{env}}\langle heap_3\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_2$
$heap_4 \equiv$ list(?x)([?v]@?B), list(?n)(?D), σ
$\psi_3 \equiv$ A = rev([?v]@?B)@?D
$\varphi_{18} \equiv \langle\langle while_1 \curvearrowright k\rangle_{\mathsf{k}}\langle env_4\rangle_{\mathsf{env}}\langle heap_4\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_3$
$\varphi_{19} \equiv \langle\langle k\rangle_{\mathsf{k}}\langle env_1\rangle_{\mathsf{env}}\langle heap_1\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \wedge \psi_1 \wedge ?x = 0$
$\theta_1 \equiv$ {?p $\mapsto$?x, ?x $\mapsto$?n, ?y $\mapsto$?n, ?B $\mapsto$ [?v]@?B, ?C $\mapsto$?D}
$X_1 \equiv$ {?p, ?x, ?y, ?B, ?C}
$X_2 \equiv$ {?v, ?n, ?D}
$\mathcal{S}_1 \equiv$ semantics of the C fragment
$\mathcal{S}_2 \equiv \mathcal{S}_1 \cup \{\varphi_1 \Rightarrow \exists X_1 \varphi_{19}\}$

1	$\mathcal{S}_1 \vdash \varphi_1 \Rightarrow^+ \varphi_2$	**ASLF**		
2	$\mathcal{S}_2 \vdash \varphi_2 \Rightarrow \varphi_3$	**ASLF**		
3	$\models \varphi_3 \rightarrow \varphi_4 \vee \varphi_5$	$\text{FOL}_=$		
4	$\mathcal{S}_2 \vdash \varphi_5 \Rightarrow \varphi_7$	**ASLF**		
5	$\mathcal{S}_2 \vdash \varphi_7 \Rightarrow \varphi_{19}$	**ASLF**		
6	$\models \varphi_{19} \rightarrow \exists X_1 \varphi_{19}$	$\text{FOL}_=$		
7	$\mathcal{S}_2 \vdash \varphi_7 \Rightarrow \exists X_1 \varphi_{19}$	**Consequence**	5, 6	
8	$\mathcal{S}_2 \vdash \varphi_5 \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	4, 7	
9	$\mathcal{S}_2 \vdash \varphi_4 \Rightarrow \varphi_6$	**ASLF**		
10	$\models \varphi_6 \rightarrow \exists X_2 \varphi_8$	$\text{FOL}_=$	*list*	
11	$\mathcal{S}_2 \vdash \varphi_8 \Rightarrow \varphi_9$	**ASLF**		
12	$\mathcal{S}_2 \vdash \varphi_9 \Rightarrow \varphi_{10}$	**ASLF**		
13	$\mathcal{S}_2 \vdash \varphi_{10} \Rightarrow \varphi_{11}$	**ASLF**		
14	$\mathcal{S}_2 \vdash \varphi_{11} \Rightarrow \varphi_{12}$	**ASLF**		
15	$\mathcal{S}_2 \vdash \varphi_{12} \Rightarrow \varphi_{13}$	**ASLF**		
16	$\mathcal{S}_2 \vdash \varphi_{13} \Rightarrow \varphi_{14}$	**ASLF**		
17	$\mathcal{S}_2 \vdash \varphi_{14} \Rightarrow \varphi_{15}$	**ASLF**		
18	$\mathcal{S}_2 \vdash \varphi_{15} \Rightarrow \varphi_{16}$	**ASLF**		
19	$\mathcal{S}_2 \vdash \varphi_{16} \Rightarrow \varphi_{17}$	**ASLF**		
20	$\models \varphi_{17} \rightarrow \varphi_{18}$	$\text{FOL}_=$	*list*	
21	$\mathcal{S}_2 \vdash \varphi_1 \Rightarrow \exists X_1 \varphi_{19}$	**Axiom**		
22	$\mathcal{S}_2 \vdash \varphi_{18} \Rightarrow \exists X_1 \varphi_{19}$	**Substitution**	21, θ_1	
23	$\mathcal{S}_2 \vdash \varphi_{17} \Rightarrow \exists X_1 \varphi_{19}$	**Consequence**	20, 22	
24	$\mathcal{S}_2 \vdash \varphi_{16} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	19, 23	
25	$\mathcal{S}_2 \vdash \varphi_{15} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	18, 24	
26	$\mathcal{S}_2 \vdash \varphi_{14} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	17, 25	
27	$\mathcal{S}_2 \vdash \varphi_{13} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	16, 26	
28	$\mathcal{S}_2 \vdash \varphi_{12} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	15, 27	
29	$\mathcal{S}_2 \vdash \varphi_{11} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	14, 28	
30	$\mathcal{S}_2 \vdash \varphi_{10} \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	13, 29	
31	$\mathcal{S}_2 \vdash \varphi_9 \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	12, 30	
32	$\mathcal{S}_2 \vdash \varphi_8 \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	11, 31	
33	$\mathcal{S}_2 \vdash \exists X_2 \varphi_8 \Rightarrow \exists X_1 \varphi_{19}$	**Abstraction**	32	
34	$\mathcal{S}_2 \vdash \varphi_6 \Rightarrow \exists X_1 \varphi_{19}$	**Consequence**	10, 33	
35	$\mathcal{S}_2 \vdash \varphi_4 \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	9, 34	
36	$\mathcal{S}_2 \vdash \varphi_4 \vee \varphi_5 \Rightarrow \exists X_1 \varphi_{19}$	**CaseAnalysis**	8, 35	
37	$\mathcal{S}_2 \vdash \varphi_3 \Rightarrow \exists X_1 \varphi_{19}$	**Consequence**	3, 36	
38	$\mathcal{S}_2 \vdash \varphi_2 \Rightarrow \exists X_1 \varphi_{19}$	**Transitivity**	2, 37	
39	$\mathcal{S}_1 \vdash \varphi_1 \Rightarrow \exists X_1 \varphi_{19}$	**Circularity**	1, 38	
40	$\mathcal{S}_1 \vdash \exists X_1 \varphi_1 \Rightarrow \exists X_1 \varphi_{19}$	**Abstraction**	39	

Figure 11. Matching logic rewriting proof of the **`while`** loop in function `reverseList` in Figure 3