Checking Reachability using Matching Logic [*]

Grigore Roşu
University of Illinois at Urbana-Champaign
Alexandru Ioan Cuza University, Iaşi, Romania
grosu@illinois.edu

Andrei Ştefănescu
University of Illinois at Urbana-Champaign
stefane1@illinois.edu

Abstract

This paper presents a verification framework that is parametric in a (trusted) operational semantics of some programming language. The underlying proof system is language-independent and consists of eight proof rules. The proof system is proved partially correct and relatively complete (with respect to the programming language configuration model). To show its practicality, the generic framework is instantiated with a fragment of C and evaluated with encouraging results.

Categories and Subject Descriptors D.2.4 [*Software Verification*]: Formal methods; D.2.5 [*Testing and Debugging*]: Symb. execution; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

General Terms Languages, Verification, Theory

Keywords Reachability, Hoare logic, matching logic

1. Introduction

Compared to other programming language semantic approaches, operational semantics are easier to understand and define, since we can think of them as formal interpreters for the languages they define. Moreover and perhaps more importantly, they can be efficiently executable, and thus testable, the same way we test language implementations: by executing large test suites of programs (see, e.g., [1, 4, 10] for the case of C). This way, errors are detected and fixed, and confidence in the semantics is incrementally build. It is therefore quite common that operational semantics are considered trusted reference models of the programming languages they define.

On the other hand, existing program reasoning approaches, such as Hoare/separation/dynamic logics, require us to (re)define the target language semantics as a set of proof rules, which are often hard to understand and thus trust. For that reason, the state-of-the-art in mechanical verification is to prove such language-specific proof systems sound with respect to another, more trusted semantics. Since the semantics of real languages can consist of hundreds or even thousands of rules, this duplication of work is at best uneconomical. Worse, in order to even be possible to define a language semantics in these formalisms, non-trivial and specialized extensions of program logics are developed (e.g., "a separation logic for Javascript", etc.). Since such extensions are likely non-reusable, they are not worth investigating in full depth. Consequently, current program verification approaches and tools are not easy to adapt to real languages, and even the most advanced are only shown sound and/or relatively complete for small fragments of their target languages.

What we want is a framework which takes a trusted semantics of some arbitrary programming language, say an off-the-shelf well-tested operational semantics, and automatically (with zero effort) provides the following: (1) an expressive specification formalism for properties of programs in that language; and (2) a sound, complete and practical means for proving such language-specific program properties. Building upon matching logic for (1), we give the first language-independent, sound and relatively complete proof system for (2). Our proof system consists of eight proof rules for reachability. To show its practicality, we have applied the generic framework to a fragment of the C language, by simply passing the operational semantics of the language, unchanged, as axioms to the proof system, and by adding a few heuristics for which proof rules to apply and when.

Matching logic [31] is designed to state and reason about structural properties of 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. For a configuration signature with a top-level cell $\langle ... \rangle_{\mathsf{cfg}}$ holding other cells with semantic data such as code $\langle ... \rangle_{\mathsf{k}}$, environment $\langle ... \rangle_{\mathsf{env}}$, heap $\langle ... \rangle_{\mathsf{heap}}$, input buffer $\langle ... \rangle_{\mathsf{in}}$, output buffer $\langle ... \rangle_{\mathsf{out}}$, etc., configurations have the form

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

[*] The companion report [30] includes proofs of all the claimed results.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10…$10.00

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

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

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 [10] has more than 70 nested cells. However, no matter how complex a language is, its configurations can be defined as elements in an algebra, 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

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

is satisfied by all configurations where program variable x holds 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. Matching logic allows us to reason about configurations, e.g., to prove:

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

Since configuration terms with variables are particular patterns, typical reduction rules *left* $\Rightarrow$ *right* used to define operational semantics (several operational semantics approaches based on such rules are discussed in Section 5) are particular matching logic reachability rules [28] of the form $\varphi \Rightarrow \varphi'$, where φ and φ' are patterns, so such operational semantics become sets of reachability rules. For example, the following reachability rule gives the operational semantics of variable assignment in a language over configurations like above:

$$\langle\langle \texttt{x} = i\texttt{;}\,\texttt{s}\rangle_{\mathsf{k}}\ \langle \texttt{x} \mapsto j,\ e\rangle_{\mathsf{env}}\ c\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \texttt{s}\rangle_{\mathsf{k}}\ \langle \texttt{x} \mapsto i,\ e\rangle_{\mathsf{env}}\ c\rangle_{\mathsf{cfg}}$$

Reachability rules as used above only specify one-step reachability, but in general they can express any number of steps. As shown in [29], they are in fact expressive enough to specify any program properties that can be expressed using Hoare triples. Therefore, despite conceptual simplicity, such reachability rules are quite expressive, subsuming the main elements of both operational and axiomatic semantics. These observation makes it desirable to have a framework allowing to derive sequents $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$, where $\mathcal{A}$ is a set of "axioms" (i.e, trusted) reachability rules and $\varphi \Rightarrow \varphi'$ a reachability rule to be proved. If $\mathcal{A}$ is the semantics of C and $\varphi \Rightarrow \varphi'$ a reachability property of some C program, for example, then such a proof system would allow us to prove such a property based entirely on the operational semantics of C. No other semantics for verification purposes would be needed.

In this paper we give an eight-rule proof system for more general sequents, of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, where C is also a set of reachability rules. The intuition for such sequents is that the reachability property $\varphi \Rightarrow \varphi'$ holds under the hypotheses $\mathcal{A}$ and C, provided that the first step is always one from $\mathcal{A}$ whenever C is non-empty. The rules in C are called *circularities*. The desired sequents $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ are recovered when C is empty. The characteristic rule of our proof system, which allows to add circularities, is the following:

$$\textbf{Circularity}: \qquad \frac{\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$$

The Circularity rule therefore allows to make a new circularity claim at any moment during a proof derivation. In practice one typically makes such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If one succeeds to prove the circularity claim using itself, then the claim holds. This circular reasoning would obviously be unsound if used unrestricted. What makes the reasoning with circularities well-founded and thus sound is the following modified Transitivity rule, which unleashes the circularities only after at least one trusted step, i.e., one from the set of axioms $\mathcal{A}$ (i.e., an operational semantics rule), is applied:

$$\textbf{Transitivity}: \qquad \frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow^+ \varphi_2 \qquad \mathcal{A} \cup C \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

The $\Rightarrow^+$ in the first premise implies at least one step corresponding to a rule in $\mathcal{A}$ has been applied. In concrete instances, that operational step is typically a loop unrolling step, or a function invocation step, or a jump, etc. The remaining six proof rules (Figure 7) and the soundness theorem for the entire proof system (Theorem 1), in the sense of partial correctness, are given in Section 6.

Our language-independent eight-rule proof system can therefore be used to derive reachability properties of programs in any language, provided that an operational semantics of that language is provided. But how powerful is it? In Section 7 we show that our proof system is in fact relatively complete. That means that it can derive any reachability property of any program in the given programming language. Note that this is a significant improvement over the state-of-the-art, as conventional program verification logics, like Hoare logic or dynamic logic, need to be proved relatively complete for each language separately. We prove our relative completeness result for matching logic reachability once and for all, for all languages. The relativity in our completeness result comes from the fact that our setting, including the proof system, are parametric in a model of configurations. We assume an oracle telling whether the arbitrary but fixed configuration model satisfies a given first-order formula or not.

To test the effectiveness of our approach, we implemented a program verifier for a fragment of C, called MatchC, which is directly based on the proof system in Figure 7. It uses the operational semantics of the fragment of C completely unchanged for program verification. A series of non-trivial programs have been automatically verified using MatchC, some discussed in Sections 3 and 8. The Matching Logic web page, `http://fsl.cs.uiuc.edu/ml`, contains an online interface to run MatchC, where one can try more than 50 existing examples (or type your own). For example, the functional correctness of the Schorr-Waite graph marking algorithm verifies in less than 2 seconds. Section 8 also discusses how the proof system is being made practical.

Contributions

This paper makes the following specific contributions:

1. A novel language-independent proof system, parametric in an operational semantics of a programming language;
2. A proof of its partial correctness, saying that any derived reachability property is semantically valid;
3. A proof of its relative completeness, saying that any semantically valid reachability property can be derived;
4. An implementation of it in MatchC, together with several challenging C programs verified.

We also show that separation logic can be framed as an instance of matching logic for an idealized model of heaps.

2. Related Work

The idea of regarding a program (fragment) as a specification transformer to analyze programs in a forwards-style goes back to Floyd [13]. However, his rules are not concerned with 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 [14]. Evolving specifications [24] adapt and extend this basic idea to compositional systems, refinement and behavioral specifications. What distinguishes the various specification transforming approaches is the formalism they use. What distinguishes matching logic is its apparently low-level formalism, dropping no detail from the configuration. The use of variables in patterns offers a comfortable level of abstraction by mentioning in each rule only the necessary configuration components.

The state of the art in mechanized program verification [1, 21] 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 eight-rule proof system proposed in this paper is language-independent and it should be easy to mechanize in a higher-order framework.

Separation logic [22, 25] is an extension of Hoare logic. There is a major difference between separation and matching logic: 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. We study more closely the relationship to separation logic in Section 4.3.

Bedrock [5] 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. Specifications use operators defined in a pure functional language, similarly to the operators defined algebraically in matching logic. Tactics employed by Bedrock could likely be adapted for higher-order matching logic.

Shape analysis [32] allows to examine and verify properties of heap structures. The ideas of shape analysis have also been combined with those of separation logic [9] to quickly infer invariants for programs operating on lists. They can likely be also combined with matching logic to infer patterns.

Dynamic logic (DL) [15] adds modal operators to FOL to embed program fragments within specifications. For example, $\varphi \rightarrow [\mathtt{s}]\varphi'$ means "after executing `s` in a state satisfying φ, a state may be reached which satisfies φ'". In matching logic, programs and specifications also coexits in the same logic. However, matching logic achieves it by staying within FOL and making use of FOL's algebraic signatures and term models. Moreover, DL adds language-specific proof rules, while our proof system is language-independent.

Our own related work. Matching logic was introduced in [31]. However, the proof system there was language-specific and not parametric in an operational semantics like the one here. An early implementation of MatchC, based on the proof system in [31], was presented in [27]. Our first language-independent proof system is given in [28], and consists of nine rules for deriving sequents $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$, that is, ones without circularities C with them. There, the Circularity rule was more complex, mixing ideas from our current Circularity and Transitivity rules, and a now unnecessary Substitution rule was included. In [29] we were able to show the relative completeness of that proof system for a particular simple language, IMP, by showing how each Hoare logic proof rule of IMP could be mimicked with the proof system in [28]. While all the above are clearly inferior to our new proof system and its language-independent proofs of soundness and completeness, they were nevertheless crucial milestones.

3. Examples using MatchC

Here we discuss a few C examples that illustrate the expressiveness and practicality of our approach. Figure 1 shows an undefined program; Figure 2 a function that reverses a

```
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 1. C program exhibiting undefined behaviour.

singly linked list; Figure 3 a function that reads a sequence of integers from the standard input into a singly-linked list; Figure 4 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 5 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 w.r.t. their specifications (given in the grey boxes) in ~1s in total (Section 8).

The unannotated/unspecified program in Figure 1 is undefined according to the C standard: it attempts to print the value of the uninitialized list member `next`. Our operational semantics correctly captures undefinedness, in that undefined programs get stuck during their execution using the semantics. MatchC verifies programs by executing them according to the semantics. 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 reachability rules. Since this program is unannotated, its verification reduces to executing it according to 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`.

Some* MatchC *notations. Each user-supplied rule or invariant annotation (grayed area in the figures in this section) corresponds to a reachability rule, also called a specification, that needs to be derived with the proof system in Figure 7. For the next specifications, we discuss some MatchC notations that help avoid verbosity. (1) While all specifications are reachability rules $\varphi \Rightarrow \varphi'$, often φ and φ' share configuration context; we only mention the context once and distribute "$\Rightarrow$" through the context where the changes take place. (2) To avoid writing existential quantifiers, logical variables starting with "?" are assumed existentially quantified over the current pattern. (3) To avoid writing environment cells with only bindings of the form $\mathtt{x} \mapsto ?\mathsf{x}$, we automatically assume them when not explicitly mentioned and allow users to write the identifier `x` (i.e., a syntactic constant) instead of the logical variable $?\mathsf{x}$. (4) MatchC desugars invariants `inv` φ `loop` into rules $\varphi[\mathtt{loop...}] \Rightarrow \varphi[\mathtt{...}] \wedge \neg cond(\mathtt{loop})$, with $\varphi[\mathtt{code}]$ the pattern obtained from φ setting the contents of $\langle ... \rangle_{\mathsf{k}}$ to `code`.

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

struct listNode* reverseList(struct listNode *x)
```

rule $\langle \$ \Rightarrow \textbf{return}\ ?\mathsf{p};\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \mathsf{list}(\mathtt{x})(\mathsf{A}) \Rightarrow \mathsf{list}(?\mathsf{p})(\mathsf{rev}(\mathsf{A}))\ \cdots\rangle_{\mathsf{heap}}$

```
{
  struct listNode *p;
  p = NULL;
```

inv $\langle\cdots\ \mathsf{list}(\mathsf{p})(?\mathsf{B}),\ \mathsf{list}(\mathsf{x})(?\mathsf{C})\ \cdots\rangle_{\mathsf{heap}} \wedge \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C}$

```
  while(x != NULL) {
    struct listNode *y;
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

Figure 2. C function reversing a singly-linked list.

Function `reverseList` in Figure 2 reverses a singly-linked list. The matching logic rule specifying its behavior says that it returns a pointer $?\mathsf{p}$ (here and in the rest of the paper, \$ stands for the body of the function). The rule also says that, when the function is called, the heap contains a list starting at `x` with contents the sequence A. When the function returns, the initial list is replaced by a list starting at $?\mathsf{p}$ with contents the reversed sequence, $\mathsf{rev}(\mathsf{A})$. The $\cdots$ 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, $?\mathsf{B}$, and one starting at `x` and containing the part of the sequence that is yet to be reversed, $?\mathsf{C}$. The initial sequence A equals $\mathsf{rev}(?\mathsf{B})$ followed by $?\mathsf{C}$. Again, the rest of the heap and configuration stay unchanged. Here list, rev, etc., are ordinary operation symbols in the signature and constrained through axioms (Section 4.2). Like in OCaml, @ concatenates sequences. Variables without ?, like A, are free. Hence, A refers to the same sequence in the function rule and in the loop invariant, while $?\mathsf{B}$ can refer to different sequences in different loop iterations.

One might, at this early stage, argue that separation logic allows writing more compact specifications. For example, with the same convention about ? variables, i.e., they are existentially quantified over the entire formula, the invariant in Figure 2 would be specified in separation logic as

$$(list(\mathsf{p}, ?\mathsf{B})\ *\ list(\mathsf{x}, ?\mathsf{C}))\ \wedge\ \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C},$$

where $list(\mathsf{p}, ?\mathsf{B})$ is a *predicate* capturing the same intuition as our *term* $\mathsf{list}(\mathsf{p})(?\mathsf{B})$. While this separation logic formula is indeed slightly more compact than our matching logic pattern, we would like to make two observations:

```
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 3. 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 4. C program respecting a stack inspection policy.

- First, separation logic is heap-centric in its semantics, so "$*$" automatically refers to the heap, while matching logic makes no such assumptions. If the heap were the only cell in the configuration, then we could easily adopt the assumption that heap terms are automatically wrapped within the heap cell, in which case our notation would be just as compact. However, as seen shortly, we introduce input/output buffers and a call stack to the configuration. Then the uniform notation which explicitly mentions the cells becomes quite natural and useful; separation logic would require syntactic and semantic extensions to deal with such additional components. As shown in Section 4.3, any separation logic formula can be mechanically translated into an equivalent matching logic pattern.
- Second, the compactness of separation logic formulae is also due to an implicit *heap framing rule* in Hoare logics based on separation logic. In matching logic verification we deliberately avoid adding any automatic framing rules, simply because those are not necessary. For example, the "..." symbols in the specifications in Figure 2 are anonymous (first-order) variables that match the corresponding cell "frames". Removing all the "..." from the heap cells would state that `reverseList` can only be called in contexts where the heap contains nothing but a list that `x` points to. This would be hard to specify using separation logic with implicit heap framing.

Function `readList` in Figure 3 reads `n` integers from standard input and stores them in a singly-linked list. The 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 A (replaces the empty heap ·). 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 $\mathtt{n} - \mathtt{i}$, 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 4 shows a C program that respects the following security policy: `trusted` must always be called directly with `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 reachability 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`. In particular, both `trusted` and `untrusted` require the heap to stay unchanged. We can prove that, as neither of the three functions allocates or deallocates heap memory. Function `any` does not have a rule, so its body is executed at each call. 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. Note that just constructing the call graph and performing value analysis is not enough to verify these stack properties.

Function `treeToList` in Figure 5 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, 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 says that it returns pointer ?l. The rule matches in the heap a tree rooted at `t` with contents T and replaces it with a list starting at ?l with 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), equals 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.

4. Matching Logic: A Logic of Configurations

Traditionally, program logics are deliberately not concerned with low-level details about 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

```
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 $\langle \$ \Rightarrow$ **return** ?l; $\cdots\rangle_k$ $\langle\cdots$ tree(x)(T) $\Rightarrow$ list(?l)(tree2list(T)) $\cdots\rangle_{heap}$
$\langle\cdots\ \cdot \Rightarrow$ rev(tree2list(T))$\rangle_{out}$

```
{
  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 $\langle\cdots$ tree(s)(?TS), list(l)(?A) $\cdots\rangle_{heap}$ $\langle\cdots$ rev(?A)$\rangle_{out}$
$\wedge$ 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 5. Iterative C program flattening a tree into a list and printing its values in the process.

in non-conventional ways, or sometimes both. In contrast, matching logic [31] takes program configurations at its core.

We next first recall general matching logic notions and notations (Section 4.1) with emphasis on its patterns, then give an instance of it for configurations corresponding to a fragment of the C language (Section 4.2), and in the end discuss how it relates to separation logic [23, 25] (Section 4.3).

4.1 Patterns and General Notions

Matching logic [31] is a logic suitable for specifying and reasoning about program or system configurations. Although originally framed as a methodological fragment of first-order logic (FOL), a setting that also suffices for this paper, matching logic can be easily extended to second- or higher-order settings. Matching logic is parametric in a syntax and a model for configurations. Some configurations can be as simple as pairs $\langle \texttt{code}, \sigma \rangle$ with `code` a fragment of program and σ a "state" map from program variables to integers, e.g. when one wants to reason about simple imperative languages. Other configurations can be even simpler, for example just "heap" singletons holding a map from locations to integers (e.g., when one wants to exclusively reason about heap structures like in separation logic; see Section 4.3) or even just "code" singletons (e.g., when one wants to reason about programs based purely on their syntax). Yet, other configurations can be as complex as that of the C language [10], which contains more than 70 semantic components. No matter how simple or complex the configurations under consideration are, the same machinery described below works for all.

We assume the reader is familiar with basic concepts of algebraic specification and first-order logic. Given an *algebraic signature* Σ, we let T_Σ denote the *initial* Σ*-algebra* of ground terms (i.e., terms without variables) and let $T_\Sigma(Var)$ denote the *free* Σ*-algebra* of terms with variables in Var. $T_{\Sigma,s}(Var)$ is the set of Σ-terms of sort s. Maps $\rho : Var \to \mathcal{T}$ with $\mathcal{T}$ a Σ-algebra extend uniquely to (homonymous) Σ-*algebra morphisms* $\rho : T_\Sigma(Var) \to \mathcal{T}$. These notions extend to algebraic specifications. Many mathematical structures needed for language semantics have been defined as initial Σ-algebras: boolean algebras, natural/integer/rational numbers, lists, sets, bags (or multisets), maps (e.g., for states, heaps), trees, queues, stacks, etc. We refer the reader to the CASL [20] and Maude [6] manuals for examples.

Let us fix the following: (1) an algebraic signature Σ, associated to some desired configuration syntax, with distinguished sort Cfg, (2) a sort-wise infinite set of variables Var, and (3) a Σ-algebra $\mathcal{T}$, the *configuration model*, which may but needs not necessarily be the initial or free Σ-algebra. As usual, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort Cfg, which we call *configurations*. In Section 7, we prove the relative completeness of our proof system w.r.t. such an arbitrary but fixed configuration model (with some additional constraints).

Definition 1. *[31] A matching logic formula, or a* ***pattern****, is a first-order logic (FOL) formula which allows* terms *in* $T_{\Sigma,Cfg}(Var)$*, called* ***basic patterns****, as* predicates*. We define the satisfaction* $(\gamma,\rho) \models \varphi$ *over configurations* $\gamma \in \mathcal{T}_{Cfg}$*, valuations* $\rho : Var \to \mathcal{T}$ *and patterns* φ *as follows (among the FOL constructs, we only show* $\exists$*):*

$$\begin{array}{lll}(\gamma,\rho) \models \exists X\, \varphi & \textit{iff} & (\gamma,\rho') \models \varphi \textit{ for some } \rho' : Var \to \mathcal{T} \textit{ with} \\ & & \rho'(y) = \rho(y) \textit{ for all } y \in Var \backslash X \\ (\gamma,\rho) \models \pi & \textit{iff} & \gamma = \rho(\pi) \quad \textit{, where } \pi \in T_{\Sigma,Cfg}(Var)\end{array}$$

A pattern φ *is* ***valid****, written* $\models \varphi$*, when* $(\gamma,\rho) \models \varphi$ *for all* $\gamma \in \mathcal{T}_{Cfg}$ *and all* $\rho : Var \to \mathcal{T}$*.*

A basic pattern π is satisfied by all the configurations γ that *match* it; the ρ in $(\gamma,\rho) \models \pi$ can be thought of as the "witness" of the matching, and can be further constrained in a pattern. If `SUM` is the code "`s:=0; while(n>0)(s:=s+n; n:=n-1)`" in a simple imperative language with configurations $\langle \texttt{code}, \sigma \rangle$ e.g., then the pattern $\exists s\, (\langle\ \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\ \rangle \wedge n \geq_{Int} 0)$ matches the configurations with code `SUM` and state binding program variables `s` and `n` to integers s and respectively $n \geq_{Int} 0$. We typically use `typewriter` for program variables and *italic* for mathematical variables in Var. Pattern reasoning reduces to FOL reasoning in the configuration model $\mathcal{T}$:

Definition 2. *Let* $\square$ *be a special fresh Cfg variable, which is not in Var, and let* $Var^\square$ *be the extended set of variables* $Var \cup \{\square\}$*. For a pattern* φ*, let* $\varphi^\square$ *be the FOL formula obtained by replacing basic patterns* $\pi \in T_{\Sigma,Cfg}(Var)$ *with equalities* $\square = \pi$*. If* $\gamma \in \mathcal{T}_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$*, then let* $\rho^\gamma : Var^\square \to \mathcal{T}$ *be the valuation which extends* ρ *by mapping* $\square$ *into* γ*:* $\rho^\gamma(\square) = \gamma$ *and* $\rho^\gamma(x) = \rho(x)$ *for all* $x \in Var$*. To highlight the semantic indistinguishability between matching logic patterns with variables in Var and the corresponding fragment of FOL with variables in* $Var^\square$*, we take the freedom to write* $(\gamma,\rho) \models \varphi^\square$ *in the FOL fragment, too, instead of* $\rho^\gamma \models \varphi^\square$*. A matching logic (respectively FOL) formula* ψ *is* ***patternless*** *iff it contains no basic pattern (respectively no* $\square$ *variable), that is,* $\psi = \psi^\square$*.*

The following proposition states that the notation in Definition 2 is consistent:

Proposition 1. *If* φ *is a matching logic pattern,* $\gamma \in \mathcal{T}_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$*, then* $(\gamma,\rho) \models \varphi$ *(notation in Definition 1) iff* $(\gamma,\rho) \models \varphi^\square$ *(notation in Definition 2). Also* $\models \varphi$ *iff* $\mathcal{T} \models \varphi^\square$*.*

Therefore, patterns form a methodological fragment of the FOL theory of $\mathcal{T}$, so we can use conventional theorem provers or proof assistants for pattern reasoning. It is often technically convenient to eliminate the special $\square$ variable from a FOL formula $\varphi^\square$ corresponding to a matching logic pattern φ. This can be done by replacing $\square$ with a Cfg variable $c \in Var$ (possibly which does not occur free in φ): indeed, $\varphi^\square[c/\square]$ is patternless.

Lemma 1. *If* φ *is a pattern,* $c \in Var$ *is a Cfg variable, and* $\rho : Var \to \mathcal{T}$ *a valuation, then* $(\rho(c),\rho) \models \varphi^\square$ *iff* $\rho \models \varphi^\square[c/\square]$*.*

Not all patterns are equally meaningful. For example, the pattern *true* is matched by all configurations, the pattern

$$\begin{array}{rcl}
\mathit{Id} & ::= & \text{C identifiers} \\
\mathit{Nat} & ::= & \text{domain of natural numbers (including operations)} \\
\mathit{Int} & ::= & \text{domain of integer numbers (including operations)} \\
\mathit{Type} & ::= & \texttt{int} \mid \texttt{struct}\ \mathit{Id} \mid \mathit{Type}\ \texttt{*} \\
\mathit{Code} & ::= & \text{the entire remaining syntax of the C fragment} \\
\mathit{Env} & ::= & \mathit{Map}_{\mathit{Id},\mathit{Int}} \\
\mathit{TEnv} & ::= & \mathit{Map}_{\mathit{Id},\mathit{Type}} \\
\mathit{Cell} & ::= & \langle \mathit{Map}_{\mathit{Id},\mathit{List}_{\mathit{Type}\times\mathit{Id}}} \rangle_{\mathsf{struct}} \\
 & \mid & \langle \mathit{Map}_{\mathit{Id},\mathit{List}_{\mathit{Type}\times\mathit{Id}}\times K} \rangle_{\mathsf{funs}} \\
 & \mid & \langle \mathit{Code} \rangle_{\mathsf{k}} \\
 & \mid & \langle \mathit{Env} \rangle_{\mathsf{env}} \\
 & \mid & \langle \mathit{TEnv} \rangle_{\mathsf{tenv}} \\
 & \mid & \langle \mathit{Id} \rangle_{\mathsf{fname}} \\
 & \mid & \langle \mathit{List}_{\mathit{Id}\times K\times \mathit{Env}\times \mathit{TEnv}} \rangle_{\mathsf{stack}} \\
 & \mid & \langle \mathit{Map}_{\mathit{Nat},\mathit{Int}} \rangle_{\mathsf{heap}} \\
 & \mid & \langle \mathit{List}_{\mathit{Int}} \rangle_{\mathsf{in}} \\
 & \mid & \langle \mathit{List}_{\mathit{Int}} \rangle_{\mathsf{out}} \\
\mathit{Cfg} & ::= & \langle \mathit{Bag}_{\mathit{Cell}} \rangle_{\mathsf{cfg}}
\end{array}$$

Figure 6. Sample configuration

false is matched by no configurations, some patterns are always matched by precisely one configuration γ regardless of the valuation ρ, others are sometimes by matched by some configurations for some valuations, etc. For our subsequent results, we are interested in well-definedness of patterns:

Definition 3. *A pattern φ is **weakly well-defined** iff for any valuation $\rho : Var \to \mathcal{T}$ there is some configuration $\gamma \in \mathcal{T}_{Cfg}$ such that $(\gamma, \rho) \models \varphi$, and it is **well-defined** iff γ is unique.*

For example, all basic patterns π are well-defined, while patterns of the form $\pi_1 \vee \pi_2$ are weakly well-defined. Well-defined patterns have the following property

Lemma 2. *If φ is well-defined and $c_1, c_2 \in Var$ are two Cfg variables, then $\models \varphi^{\square}[c_1/\square] \wedge \varphi^{\square}[c_2/\square] \to c_1 = c_2$.*

4.2 An Instance

Here we discuss a simple but non-trivial instance of matching logic for an idealized fragment of the C language. The reason we do not choose a trivial language is because we want to reiterate that matching logic, as well as all the notions and results presented in this paper, are totally agnostic to the language under consideration and to its complexity.

To obtain a matching logic instance, one needs to provide a syntax (as a signature Σ) and a model (as a Σ-algebra) for that language's configurations. We make use of common algebraic structures like lists, sets, bags, and maps over any sorts, including other lists, sets, etc., by simply mentioning their sorts as subscripts. 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 "␣" for their concatenation, and an infix "$\mapsto$" for building map terms.

Figure 6 shows the configuration syntax of our chosen language. We only consider integer, structure and pointer types. The sort *Code* is a generic sort for "code" and comprises the entire language syntax; thus, terms of sort *Code* 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 configuration is a term $\langle ... \rangle_{\mathsf{cfg}}$ of sort *Cfg* containing a bag of cells. In addition to $\langle ... \rangle_{\mathsf{k}}$, $\langle ... \rangle_{\mathsf{env}}$ and $\langle ... \rangle_{\mathsf{tenv}}$ holding a program fragment, an environment and a type environment, $\langle ... \rangle_{\mathsf{cfg}}$ also includes the following cells: $\langle ... \rangle_{\mathsf{struct}}$ holds the available structures as a map from data structure names to lists of typed fields; $\langle ... \rangle_{\mathsf{funs}}$ holds the available functions as a map from function names to their arguments and body; $\langle ... \rangle_{\mathsf{fname}}$ holds the name of the current function; $\langle ... \rangle_{\mathsf{stack}}$ holds the function stack as a list of frames, each containing a function name and its execution context (the remaining code, the environment and the type environment); $\langle ... \rangle_{\mathsf{heap}}$ holds the heap as a map from natural numbers (pointers) to integers (values); $\langle ... \rangle_{\mathsf{in}}$ holds the input buffer as a list of integers; and $\langle ... \rangle_{\mathsf{out}}$ holds the output buffer.

Let Σ be the algebraic signature associated to the configuration syntax discussed above (it is well-known that an algebraic signature can be associated to any context-free grammar, by associating one sort to each non-terminal and one operation symbol to each production). A Σ-algebra then gives a configuration model, namely a universe of concrete language configurations. Let us assume that $\mathcal{T}$ is such a configuration model. We do not bother to define $\mathcal{T}$ concretely, because its details are irrelevant. Note, however, that $\mathcal{T}$ must include submodels of natural and integer numbers, of maps, lists, etc. Moreover, to state properties like those in Section 8, Σ needs to contain operator symbols corresponding to lists of integer numbers and append and reverse on them, for membership testing of integers to such lists, etc. Also, to meaningfully reason about programs in our language, $\mathcal{T}$ needs to satisfy certain expected properties of these operation symbols, e.g.:

$$\begin{array}{rcl}
\mathsf{rev}(\mathsf{nil}) & = & \mathsf{nil} \\
\mathsf{rev}([\mathsf{a}]) & = & [\mathsf{a}] \\
\mathsf{rev}(\mathsf{A}_1 @ \mathsf{A}_2) & = & \mathsf{rev}(\mathsf{A}_2) @ \mathsf{rev}(\mathsf{A}_1) \\
\mathsf{in}(\mathsf{a}, \mathsf{nil}) & = & \mathsf{false} \\
\mathsf{in}(\mathsf{a}, [\mathsf{b}]) & = & (\mathsf{a} = \mathsf{b}) \\
\mathsf{in}(\mathsf{a}, \mathsf{A}_1 @ \mathsf{A}_2) & = & \mathsf{in}(\mathsf{a}, \mathsf{A}_1) \vee \mathsf{in}(\mathsf{a}, \mathsf{A}_2) \\
\langle \mathsf{n}_1 \mapsto i_1, \mathsf{n}_2 \mapsto i_2, \sigma \rangle_{\mathsf{heap}} & \to & \mathsf{n}_1 \neq \mathsf{n}_2
\end{array}$$

We next give some examples of patterns for our Σ. Given program variable $\mathtt{x}$ (i.e., a constant of sort *Id*), the pattern

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

specifies those program configurations in which $\mathtt{x}$ is bound to 5 in the environment. Similarly, the pattern

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

specifies the configurations where $\mathtt{x}$ is bound to a positive integer. The next says that $\mathtt{x}$ is bound to an allocated location

$$\begin{array}{l}
\exists c{:}Bag_{Cell},\ e{:}Env,\ p{:}Nat,\ \ i{:}Int,\ \sigma{:}Map_{Nat,Int} \\
\quad \langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}
\end{array}$$

while the pattern

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

says that the location x is bound 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, we adopt the following notational conventions:

Notation 1. *Variables starting with a "?" are assumed existentially quantified over the entire pattern and thus need not be declared. The sorts of variables are inferred from their use context. Existentially quantified variables which appear only once in the pattern are 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 FOL formula, we may write $\langle e\rangle_{\mathsf{env}} \wedge \psi$ instead of $\langle\langle e\rangle_{\mathsf{env}}\ ...\rangle_{\mathsf{cfg}} \wedge \psi$.*

With these notational conventions, the patterns above become:

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

We further illustrate the expressiveness of matching logic with a few more pattern examples. The next says that program variables x and y are aliased and point to an existing location:

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

The following patterns specify configurations where program variable x is bound to the last integer that has been output (located 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:

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

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

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

The following pattern is more complex:

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

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

Assuming that γ is a configuration of $\mathcal{T}$ of the form $\langle\langle \mathtt{x} \mapsto 5,\ \mathtt{y} \mapsto 5\rangle_{\mathsf{env}}\ \langle 5 \mapsto 7\rangle_{\mathsf{heap}}\ \langle 3, 5\rangle_{\mathsf{out}}\ ...\rangle_{\mathsf{cfg}}$, then γ matches all the following patterns:

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

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

In addition to usual FOL abstractions, matching logic also allows us to introduce and axiomatize situations of interest as operations (instead of predicates). For example, we next show the list heap abstraction (part of the MatchC library) which was used, together with other similar abstractions, to verify the programs in Section 8. 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.

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

One can now use this axiom to perform reasoning like below:

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

4.3 Relationship to Separation Logic

Separation logic [23, 25] is a popular choice for specifying heap properties. Its main strength is the separation conjunction "$*$", which allows for modular reasoning. Although matching logic is not particularly concerned with specifying heap properties, the previous section showed many such properties and thus begs for a formal relationship between separation logic and matching logic. Here we present an instance of matching logic, for a particular heap-centric configuration signature and model, together with a mechanical translation of separation logic formulae into semantically equivalent patterns in the matching logic instance. There are many variations of separation logic. Here we consider first-order separation logic over integers, as presented in [23], but we believe that similar embeddings can be obtained for other variants. Formally, separation logic extends the first-order theory of integers with the following constructs:

- *emp*, the atomic predicate specifying the empty heap.
- $t_1 \mapsto t_2$, the atomic predicate specifying the singleton heap mapping the natural number (thought of as a memory location) represented by t_1 into the integer number represented by t_2.
- $P_1 * P_2$, the formula specifying the separation conjunction of two formulae, that is, that the heap can be split into two disjoint heaps satisfying P_1 and respectively P_2.

For simplicity, we do not consider the separation implication $P_1 \mathrel{-\!\!*} P_2$ here. The satisfaction of a separation logic formula P is given over a valuation s of the variables in P and a heap h, i.e., a partial function from naturals to integers. Specifically, as in [23, 25], the satisfaction of "spatial" formulae depends on both s and h, while that of "pure" formulae depends only on s (that is, is independent of h).

To establish the relation between separation logic and matching logic, for the remaining of this subsection we fix the following signature Σ with five sorts and only one cell:

$$\begin{array}{rl}
\mathit{Nat} ::= & \text{domain of natural numbers (including operations)} \\
\mathit{Int} ::= & \text{domain of integer numbers (including operations)} \\
\mathit{Bool} ::= & \text{domain of Booleans} \\
\mathit{Heap} ::= & \mathit{Map}_{\mathit{Nat},\mathit{Int}} \text{ (domain of heaps represented as finite} \\
& \text{mappings from naturals into integers)} \\
\mathit{Cfg} ::= & \langle \mathit{Heap} \rangle_{\mathsf{heap}}
\end{array}$$

As in the previous section, we use "," for the map concatenation and "." for the map unit. To Σ we associate a model $\mathcal{T}$ consisting of a model of natural numbers, a model of the integer numbers, a model of heaps, and a model of configurations (which are just heaps wrapped into a cell). We assume there is a special element $\perp$ in $\mathcal{T}$ standing for "error". All operations with at least one argument $\perp$ evaluate to $\perp$. Equality between $\perp$ and any other element does not hold. Valuations do not take any variables into $\perp$. The model of heaps has the important property that the concatenation of two maps with non-disjoint domains is $\perp$. Since separation logic cannot quantify over heap variables, if $\rho : \mathit{Var} \to \mathcal{T}$ is a valuation then we let $\overline{\rho}$ be the restriction of ρ to natural and integer variables.

Let σ, σ', σ_1, σ_2, ... be *Heap* variables in *Var* which do not occur in P. Given a separation logic formula P, we construct an equivalent matching logic formula over Σ

$$\mathit{S2M}(P) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \psi)$$

where ψ is a patternless formula. Intuitively, σ stands for the heap which ψ constrains. The construction is based on the syntactic structure of P, and somewhat mimics the definition of satisfaction for separation logic formulae:

- $\mathit{S2M}(\forall xP)$: let $\mathit{S2M}(P)$ be $\exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \psi)$. Then
$$\mathit{S2M}(\forall xP) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \forall x\, \psi)$$
- $\mathit{S2M}(P_1 \to P_2)$: let $\mathit{S2M}(P_1)$ be $\exists\sigma_1(\langle\sigma_1\rangle_{\mathsf{heap}} \wedge \psi_1)$ and $\mathit{S2M}(P_2)$ be $\exists\sigma_2(\langle\sigma_2\rangle_{\mathsf{heap}} \wedge \psi_2)$. Then we define
$$\begin{aligned}&\mathit{S2M}(P_1 \to P_2)\\ &\equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \exists\sigma_1\exists\sigma_2(\sigma = \sigma_1 \wedge \sigma = \sigma_2 \wedge (\psi_1 \to \psi_2)))\end{aligned}$$
Notice that the equalities $\sigma = \sigma_1$ and $\sigma = \sigma_2$ ensure that ψ_1 and ψ_2 constrain the same heap.
- $\mathit{S2M}(p(t_1, \ldots, t_n))$, where p is a "pure" predicate (one which is not interpreted over the heap, like "=" or "≤"):
$$\mathit{S2M}(p(t_1, \ldots, t_n)) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge p(t_1, \ldots, t_n))$$
We used the same notation for the pure predicate and the corresponding Boolean algebraic operator.
- $\mathit{S2M}(\mathit{false})$: we define
$$\mathit{S2M}(\mathit{false}) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \mathit{false})$$
- $\mathit{S2M}(P_1 * P_2)$: let $\mathit{S2M}(P_1)$ be $\exists\sigma_1(\langle\sigma_1\rangle_{\mathsf{heap}} \wedge \psi_1)$ and $\mathit{S2M}(P_2)$ be $\exists\sigma_2(\langle\sigma_2\rangle_{\mathsf{heap}} \wedge \psi_2)$. Then we define
$$\begin{aligned}&\mathit{S2M}(P_1 * P_2)\\ &\equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \exists\sigma_1\exists\sigma_2(\sigma = (\sigma_1, \sigma_2) \wedge \psi_1 \wedge \psi_2))\end{aligned}$$
Note that the equality $\sigma = (\sigma_1, \sigma_2)$ holds in $\mathcal{T}$ under some valuation ρ only if $(\rho(\sigma_1), \rho(\sigma_2))$ is a proper heap, that is, only if the domains of $\rho(\sigma_1)$ and $\rho(\sigma_2)$ are disjoint.
- $\mathit{S2M}(x \mapsto y)$: we define
$$\mathit{S2M}(t_1 \mapsto t_2) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \sigma = (t_1 \mapsto t_2))$$
We use the same notation for the separation logic predicate $t_1 \mapsto t_2$ and the algebraic map constructor $t_1 \mapsto t_2$.
- $\mathit{S2M}(\mathit{emp})$: we define (recall that $\cdot$ is the map unit)
$$\mathit{S2M}(\mathit{emp}) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \sigma = \cdot)$$

To illustrate the transformation, consider the separation logic formula $P = x \mapsto a * y \mapsto b \wedge a \neq b$. By applying the transformation we have $\mathit{S2M}(P)$ to be

$$\begin{aligned}&\exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \exists\sigma_1\exists\sigma_2(\sigma = \sigma_1 \wedge \sigma = \sigma_2 \wedge a \neq b\\ &\quad\wedge \exists\sigma_3\exists\sigma_4(\sigma_1 = (\sigma_3, \sigma_4) \wedge \sigma_3 = (x \mapsto a) \wedge \sigma_4 = (y \mapsto b))))\end{aligned}$$

However, after eliminating the existential quantifiers via substitution, we obtain the equivalent matching logic formula

$$\langle x \mapsto a, y \mapsto b\rangle_{\mathsf{heap}} \wedge a \neq b$$

For this reason, in practice we do not encourage the use of the transformation for generating matching logic formulae, but rather directly writing the matching logic formulae.

The following proposition formally captures the relationship between the version of separation logic considered here and the matching logic over Σ and $\mathcal{T}$.

Proposition 2. *If P is a separation logic formula, $h \in \mathcal{T}_{\mathit{Heap}}$ is a heap and $\rho : \mathit{Var} \to \mathcal{T}$ is a valuation, then $(\overline{\rho}, h) \models P$ (in separation logic) iff $(\langle h\rangle_{\mathsf{heap}}, \rho) \models \mathit{S2M}(P)$ (in matching logic). Consequently, $\models P$ (in separation logic) iff $\models \mathit{S2M}(P)$ (in matching logic).*

Although insightful, the result above is not surprising. Indeed, matching logic has the luxury of instantiating itself with any configuration signature and any model of configurations, in particular with ones that capture the precise syntax and semantics of heaps, while separation logic and its variations come with fixed such signatures and models. Therefore, the main conceptual difference between separation logic and matching logic is that the former achieves separation by means of special logical connectives and appropriate mathematical domains to interpret those, while the latter achieves separation by structural means, at the level of terms instead of modifying the logic, but with the help of an appropriately defined model of configurations. Matching logic thus has the advantage that we do not need to modify the underlying logic with each language extension that requires new semantic components to be added to the configuration, but that does not come for free: one still has to carefully construct one's configuration model with the desired properties.

5. Reachability Rules

In Section 4 we showed how one matching logic pattern φ specifies all the configurations γ that match it. Here we extend specifications to pairs of patterns, called *reachability rules* and written $\varphi \Rightarrow \varphi'$, which specify all pairs of configurations that simultaneously match both patterns. The two configurations in each such pair can be thought of as being related by the reachability relation in the transition system corresponding to the (operational semantics of the) language under consideration. Moreover, the transition system itself can be defined in terms of reachability rules, which are interpreted as one-step transitions. Reduction semantics [34] and rewriting logic [19] are perhaps the most established logical formalisms whose basic sequents are statements of the form $t \Rightarrow t'$, where t and t' are terms with variables specifying one-step transitions or reachability in the transition system associated to a reduction or rewrite system. Since matching logic patterns extend terms with logical constraints, such pairs of unconstrained terms are therefore special instances of reachability rules. We next recall notions related to reachability rules from [28, 29], and add a few new ones needed to state and prove our new soundness and completeness results.

Definition 4. *A **reachability rule** is a pair $\varphi \Rightarrow \varphi'$, where φ and φ' are patterns (which can have free variables). A **reachability system** is a set of reachability rules. A reachability 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 $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is a $\varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and a $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$.*

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. One of the most popular approaches is reduction semantics with evaluation contexts [11, 12], 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 [3], where l is a chemical solution that reacts into r under condition b. The rewriting logic semantics framework $\mathbb{K}$ [26] 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 [4, 16], and it is technically the most powerful of all the above. While we currently limit ourselves to FOL, there is nothing to prevent higher-order extensions of matching logic. Tools, techniques and methodologies supporting such operational semantics, like Redex, Maude and the $\mathbb{K}$ tool among others, as well as large languages defined using these (e.g., the C semantics [10] exceeds 1,000 such rules), stand as proof that this is not only possible, but also practical. Such rules can all be expressed as matching logic reachability rules $l \wedge b \Rightarrow r$, allowing to regard operational semantics following these approaches as matching logic reachability systems.

Generic, language-independent notions of termination and finite-branching are needed for the partial correctness and for the relative completeness results, respectively:

Definition 5. *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 starting with γ. Configuration $\gamma \in \mathcal{T}_{Cfg}$ is **finite-branching** iff the set $\{\gamma' \mid \gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'\}$ is finite. Reachability system $\mathcal{S}$ is **finite-branching** iff so is each configuration $\gamma \in \mathcal{T}_{Cfg}$.*

Finite-branching will be critical for proving the relative completeness of our proof system, since it will allow us to encode non-termination as a FOL predicate: a configuration γ does not terminate iff for any n there exists of path of length n starting with γ. An example of an infinite-branching rule is one defining a random expression construct with a reduction rule of the form $\langle \mathsf{random} \rangle \Rightarrow \langle n \rangle$ (assume a trivial language whose configuration holds only an expression) where n is a variable ranging over an idealistic (infinite) domain of natural numbers. One could devise criteria that guarantee finite-branching, such as allowing fresh variables in the right-hand sides (RHS) of rules (i.e., ones which do not appear in the rule's LHS) only if they range over finite domains, etc., but these are beyond the scope of this paper.

Definition 6. *A reachability rule $\varphi \Rightarrow \varphi'$ is **weakly well-defined**, respectively **well-defined**, iff φ' is weakly well-defined, respectively well-defined (recall Definition 3). Reachability system $\mathcal{S}$ is **(weakly) well-defined** iff each rule is (weakly) well-defined, and is **finite-branching** iff so is each configuration $\gamma \in \mathcal{T}_{Cfg}$.*

Operational semantics defined with rules "$l \Rightarrow r$ if b" are particular well-defined reachability systems with rules of the form $l \wedge b \Rightarrow r$, because r is a basic pattern and basic patterns are well-defined. One example of a properly weakly well-defined rule is one of the form $\varphi \Rightarrow \varphi_1 \vee \varphi_2$, where $(\gamma_1, \rho) \models \varphi_1$ and $(\gamma_2, \rho) \models \varphi_2$ for two different configurations γ_1 and γ_2 (however, note that such disjunctive rules can be replaced with two rules). An example of a rule $\varphi \Rightarrow \varphi'$ which is not weakly well-defined is one where φ' is not satisfiable, for example $\varphi' \equiv \mathit{false}$. Such non-well-defined rules are unlikely to appear in any meaningful operational semantics, but nevertheless, we do not want to impose any particular style or methodology to define operational semantics in this paper and instead prefer to prove our generic soundness and completeness results as generally as possible. Weak well-definedness will be required for the soundness of our proof system and well-definedness for our completeness result.

Note that there is no relationships between finite branching and well-definedness. For example, the rule $\langle \mathsf{random} \rangle \Rightarrow \langle n \rangle$ above is well-defined but not finite branching, while the rules $\varphi \Rightarrow \varphi_1 \vee \varphi_2$ and $\varphi \Rightarrow \mathit{false}$ are finite branching but not well-defined (the latter is not even weakly well defined).

Reachability rules can specify not only operational semantics of languages, but also program properties. In fact, each Hoare triple can be regarded as a particular reachability rule [29], although the translation needs to be mechanized separately for each language. For example, the property of the SUM program mentioned in Section 4.1 in the context of a simple imperative language with configurations of the form $\langle \texttt{code}, \sigma \rangle$ would be

$$\begin{array}{l}\exists s\,(\langle \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \wedge n \geq_{Int} 0) \\ \Rightarrow\ \langle \texttt{skip},\ (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2,\ \texttt{n} \mapsto 0)\rangle\end{array}$$

Unlike Hoare triples, which only specify properties about final program states, reachability rules can also specify intermediate state properties. Hoare triples correspond to reachability rules whose basic right-hand pattern holds the empty code, like the one above. Semantic validity in matching logic captures the same intuition of *partial correctness* as Hoare logic, but in more general terms of reachability:

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

If φ' holds the empty code, then so does γ' in the definition above, and in that case γ' is unique and thus we recover the Hoare validity as a special case.

6. Language-Independent Proof System

Figure 7 shows our language-independent matching logic proof system for reachability. It derives sequents of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, where $\mathcal{A}$ and C are sets of reachability rules. Initially, $\mathcal{A}$ contains the operational semantics of the target language, given as a set of (one-step) reachability rules, and C is empty. We call the rules in C *circularities*. When C is empty, we write the sequent as $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$. The intuition for a sequent $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$ is that the reachability rule $\varphi \Rightarrow \varphi'$ holds under the hypotheses $\mathcal{A}$ and C, provided that the first step is always one from $\mathcal{A}$ whenever C is non-empty. In other words, the existence of a (non-empty) C in a sequent $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$ requires its derivation to start with a trusted step, that is, a step corresponding to a trusted rule in $\mathcal{A}$, and then to continue unrestricted using steps from both $\mathcal{A}$ and C.

The proof rules Axiom, Reflexivity, and Transitivity have an operational nature and their role is mainly to (symbolically) execute operational semantics. Note how they properly capture the intuition of our sequents: Axiom initiates the trusted steps, Reflexivity is only allowed when C is empty, and Transitivity requires at least one first trusted step with axioms in $\mathcal{A}$, followed by unrestricted use of rules in both $\mathcal{A}$ and C as axioms. Logic Framing allows the deduction of reachability to take place in context, but only when the context is patternless. In other words, it is safe to add more logical constraints on existing reachability properties, but it is unsafe to add more structural constraints. We let it as an exercise to the reader to note why it would be unsafe to allow structural constraints in frames. Consequence, Case Analysis and Abstraction are reminiscent to homonymous proof rules that appear in Hoare logic proof systems. The latter two can typically be proved in Hoare logics, by structural induction on the language syntax, but they are necessary in our language-independent system.

The Circularity proof rule has an inductive/coinductive nature and captures the various circular behaviors that appear in languages, due to loops, recursion, etc. Circularity allows us to make a claim of circular behavior at any moment during a proof derivation. The claim holds if we succeed to prove it ... using itself. What makes such a reasoning well-founded and thus sound is the fact that the circularity claim will only be allowed to be used after at least one trusted step. In concrete instances, that trusted step is typically a loop unrolling, or a function invocation, or a jump, etc., as given by the operational semantics of the language.

The proof system in Figure 7 is clearly agnostic to the particular operational approach or style used to define the target language. In Section 8 we describe our particular

Axiom :

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

Reflexivity :

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

Transitivity :

$$\frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow^+ \varphi_2 \qquad \mathcal{A} \cup C \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

Logic Framing :

$$\frac{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \qquad \psi \text{ is a (patternless) FOL formula}}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence :

$$\frac{\models \varphi_1 \to \varphi_1' \qquad \mathcal{A} \vdash_C \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \to \varphi_2}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

Case Analysis :

$$\frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_C \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Abstraction :

$$\frac{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \qquad X \cap FreeVars(\varphi') = \emptyset}{\mathcal{A} \vdash_C \exists X\, \varphi \Rightarrow \varphi'}$$

Circularity :

$$\frac{\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$$

Figure 7. Matching logic proof system for reachability (eight language-independent proof rules). The use of $\Rightarrow^+$ in sequent means that it was derived without Reflexivity.

```
⟨
 ⟨··· x ↦ ?x, p ↦ ?p, y ↦ ?y ···⟩env
 ⟨··· list(?p)(?B), list(?x)(?C) ···⟩heap
 ⟨ if(x != NULL) {
     y = x->next;
     x->next = p;
     p = x;
     x = y;
   } ···⟩k
 ···
⟩cfg ∧ A = rev(?B)@?C
⇒
⟨
 ⟨··· x ↦ ?x, p ↦ ?p, y ↦ ?y ···⟩env
 ⟨··· list(?p)(?B), list(?x)(?C) ···⟩heap
 ⟨···⟩k
 ···
⟩cfg ∧ A = rev(?B)@?C
```

Figure 8. Matching logic reachability rule derivable with the first seven rules of the proof system in Figure 7 with $\mathcal{S}$ the operational semantics of the considered fragment of C. The ellipses in each pattern stand for distinct free variables, assumed the same on the corresponding positions in the left-hand-side and the right-hand-side.The "?" variables are existentially quantified over each pattern.

use of the proof system in the MatchC program verifier, using a $\mathbb{K}$ semantics of the C fragment, highlighting both its expressiveness and its potential for automation in a non-trivial language instance. Other language instances are certainly needed in order to validate the effectiveness of our proof system in other language paradigms. In particular, we are investigating its use in the context of a substitution-based reduction semantics definition of a functional language; the proof derivations appear to be simpler than for the language considered here. These results will be reported elsewhere.

6.1 Derived Proof Rules

In this section we give several derived rules that turned out to be useful in practice or in proofs. To save space, we formulate them as lemmas instead of as proof rules. The most important one is Lemma 4 (Substitution).

Lemma 3. *If $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$ then $\mathcal{A} \vdash_C \exists X\,\varphi \Rightarrow \exists X\,\varphi'$.*

Lemma 4. ***(Substitution)*** *If $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$ is derivable and $\theta : Var \to \mathcal{T}_\Sigma(Var)$, then $\mathcal{A} \vdash_C \theta(\varphi) \Rightarrow \theta(\varphi')$ is also derivable.*

Lemma 5. *Suppose that $\mathcal{A} \subseteq \mathcal{A}'$. If $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$ then $\mathcal{A}' \vdash_C \varphi \Rightarrow \varphi'$, and if $\mathcal{A} \vdash_C \varphi \Rightarrow^+ \varphi'$ then $\mathcal{A}' \vdash_C \varphi \Rightarrow^+ \varphi'$.*

Lemma 6. *If $\mathcal{A} \vdash_C \varphi \Rightarrow^+ \varphi'$ and $C \subseteq C'$ then $\mathcal{A} \vdash_{C'} \varphi \Rightarrow^+ \varphi'$.*

6.2 Examples

Here we show how the proof system in Figure 7 allows us to derive program properties expressed as reachability rules.

First, let us consider the first seven proof rules and the Substitution proof rule (which is derived from the first seven). We can use them for symbolic execution/reasoning with the operational semantics of the language. However, as seen in Example 1, they are not sufficient to derive circular behaviors.

Example 1. Let us consider the `while` loop in function `reverseList` discussed earlier (Figure 2), 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 reachability rule in Figure 8, say $\exists X\varphi \Rightarrow \exists X\varphi'$, where $X = \{?x, ?p, ?p, ?B, ?C\}$. 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 2 because of two MatchC notations: first, MatchC desugars invariants `inv` φ `loop` into matching logic rules φ[`loop`...] $\Rightarrow \varphi[...] \wedge \neg cond($`loop`$)$ (see Section 8), where φ[`code`] is the pattern obtained from φ by setting the contents of $\langle ... \rangle_k$ to `code`; second, as explained in Section 3, MatchC allows to refer directly to program variable `x` instead of logical variable ?x, generating automatically the environment cell containing bindings of the form `x` $\mapsto$?x. Since here we want to illustrate a formal proof, we completely desugar the MatchC notation in Figure 2. We first derive $\varphi \Rightarrow \exists X\varphi'$ and then the desired rule follows by Abstraction. It suffices to derive $\varphi \wedge ?x = 0 \Rightarrow \exists X\varphi'$ and $\varphi \wedge ?x \neq 0 \Rightarrow \exists X\varphi'$, and then use Case analysis and Consequence to derive $\varphi \Rightarrow \exists X\varphi'$.

- For the former, we iteratively use the semantic rules of the considered fragment of C via Axiom, Substitution, and Logical Framing, together with 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 $\varphi \wedge ?x = 0 \Rightarrow \varphi' \wedge ?x = 0$. The derivation of $\varphi \wedge ?x = 0 \Rightarrow \exists X\varphi'$ follows via Consequence, since $\models \varphi' \wedge ?x = 0 \to \exists X\varphi'$. Here we are deliberately agnostic to how the semantic rules of the language are defined, to avoid bias for any particular operational semantics approach. In Section 8 we discuss our MatchC implementation, which uses $\mathbb{K}$ [26].
- For the latter, we also use Axiom, Substitution, Logical Framing, Consequence and Transitivity until the condition of `if` evaluates to $?x \neq 0$, 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 4.2 from left-to-right; FOL reasoning eliminates the case when the list is empty (since $?x \neq 0$). Then Abstraction allows us to assume fresh variables a, q and β like in the axiom of lists and thus we can continue the execution. After the block terminates, we get:

```
⟨
 ⟨··· x ↦ q, p ↦ ?x, y ↦ q ···⟩env
 ⟨··· list(?p)(?B), ?x ↦ [a, ?p], list(q)(β) ···⟩heap
 ⟨···⟩k
 ···
⟩cfg ∧ ?C = [a]@β ∧ A = rev(?B)@?C
```

Let φ'' denote this pattern. We can now again use 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 4.2), and 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 operational semantics of the `while` loop, 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 $\langle ... \rangle_k$ 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, $\exists X\varphi \Rightarrow \exists X\varphi'$.

Next we show how the last rule of the proof system in Figure 7 can be used to deal with circular behaviors. The Circularity proof rule allows for a rule to be used in its own derivation, with certain restrictions. Initially, Circularity adds the rule to a pending set C. Typically, one or more applications of Circularity are eventually followed by Transitivity, which enables the use of the rules in C after at least one step is performed with the already existing axioms. Intuitively, Circularity allows a rule to be used in its own derivation only after progress has been made without it.

Example 2. We show how the Circularity rule can be used to verify the `while` loop in function `reverseList` in Figure 2. In Example 1, we showed how the first seven rules of the proof system in Figure 7 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. Then we argued that they cannot derive the desired property about the `while` loop, essentially 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 reachability rule in Figure 8 with `if` modified into `while` and with the extra condition ?x = 0 in the right-hand-side. Since φ contains the `while` loop in its $\langle ... \rangle_k$ cell, we can use the loop unrolling semantics and reduce φ into a pattern that resembles the left-hand-side 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 $\mathcal{S} \vdash \exists X\varphi \Rightarrow^{+} \exists X\varphi_{\texttt{if}}$. By Circularity followed by Transitivity, it suffices to derive $\mathcal{S} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \vdash \exists X\varphi_{\texttt{if}} \Rightarrow \exists X\varphi'$. Using a reasoning sequence similar to the one in Example 1, we can first derive $\varphi_{\texttt{if}} \wedge ?\mathsf{x} = 0 \Rightarrow \exists X\varphi'$ and then $\varphi_{\texttt{if}} \wedge ?\mathsf{x} \neq 0 \Rightarrow \exists X\varphi$. We next use Transitivity and Axiom with $\exists X\varphi \Rightarrow \exists X\varphi'$ to derive $\mathcal{S} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \vdash \varphi_{\texttt{if}} \wedge ?\mathsf{x} \neq 0 \Rightarrow \exists X\varphi'$, then Case analysis to derive $\mathcal{S} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \vdash \varphi_{\texttt{if}} \Rightarrow \exists X\varphi'$, and then Abstraction to derive $\mathcal{S} \cup \{\exists X\varphi \Rightarrow \exists X\varphi'\} \vdash \exists X\varphi_{\texttt{if}} \Rightarrow \exists X\varphi'$.

6.3 Partial Correctness

We now state the soundness of our proof system:

Theorem 1. ***(soundness)*** *Let $\mathcal{S}$ be a weakly well-defined reachability system and let $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ be a sequent derived with the proof system in Figure 7. Then $\mathcal{S} \models \varphi \Rightarrow \varphi'$.*

Therefore, once all the circularities are discharged, the derived reachability rule is semantically valid, in the sense of partial correctness (Definition 7). Note that this result only requires that the operational semantics of the target language is weakly well-defined, a property which is satisfied by all operational semantics that we are aware of.

Theorem 1 thus allows to formally derive correct properties of programs based entirely on the operational semantics of the language, and to produce corresponding formal certificates under the form of proof objects, without any additional axiomatic or any other kind of semantics being necessary, and without any tedious, low-level inductive proofs over the transition system associated to the operational semantics.

7. Relative Completeness

In this section we show that our proof system for reachability in Figure 6 is relatively complete. That means that any reachability property of any program in any (operational semantics of any) programming language can be formally derived with the eight proof rules of our system. The relativity comes from the fact that all our setting, including the proof system, are parametric in a configuration model. In particular, the Consequence rule in Figure 6 relies on FOL validity in the configuration model. Thus, the completeness of our proof system is relative to an oracle telling whether the configuration model satisfies a given FOL formula or not.

Before we proceed, let us note that our proof system cannot be complete without some additional constraints on the original reachability system $\mathcal{S}$. First, note that although $\mathcal{S}$ was allowed to be infinite in the partial correctness result (Theorem 1), we need to restrict it to be finite in order to prove the completeness of our proof system. Consider, for example, a setting where configurations hold natural numbers, that is, $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$, ..., and where $\mathcal{S}$ contains the infinite (although recursively enumerable) set of rules $\langle 1 \rangle \Rightarrow \langle 0 \rangle$, $\langle 2 \rangle \Rightarrow \langle 0 \rangle$, and so on. Then it is easy to see that $\mathcal{S} \models \langle n \rangle \Rightarrow \langle 0 \rangle$, where n is a variable ranging over natural numbers, but there is no way to derive $\mathcal{S} \vdash \langle n \rangle \Rightarrow \langle 0 \rangle$. Thus, $\mathcal{S}$ is restricted to be finite. This restriction is of little practical concern in our view, since operational semantics are expected to be finite, no matter how complex the target language is.

More restrictions on $\mathcal{S}$ are needed. For example, if $\mathcal{S}$ consists of the rule *true* $\Rightarrow$ *true* than the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ contains all possible pairs of configurations, so $\mathcal{S} \models \varphi \Rightarrow \varphi'$ for all reachability rules $\varphi \Rightarrow \varphi'$. On the other hand, $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ is derivable only when $\models \varphi \rightarrow \varphi'$. Thus, we need to impose further restrictions on $\mathcal{S}$ in order for our proof system to be complete. One could argue that the problem with the rule *true* $\Rightarrow$ *true* is that it does not use any basic patterns, so it is not finite branching. Consider instead a finite-branching system consisting only of a rule $\pi \Rightarrow \pi_1 \vee \pi_2$, where π, π_1, π_2 are distinct and ground basic patterns, and whose model of configurations contains precisely these three configurations. Then it is easy to see that $\mathcal{S} \models \pi \Rightarrow \pi_1$, since π_1 matches the pattern $\pi_1 \vee \pi_2$, but there is no way to derive $\mathcal{S} \vdash \pi \Rightarrow \pi_1$. Thus, finite branching does not suffice. However, the rule $\pi \Rightarrow \pi_1 \vee \pi_2$ is not well-defined. We are going to require that $\mathcal{S}$ is both finite-branching and well-defined. We believe that the finite branching requirement can be avoided, but were not able to prove the completeness result without it.

In order to formulate the FOL questions that the configuration model needs to be asked during the completeness proof, we also need some minimal support from the signature and the model of configurations. Specifically, in order to Gödelize over sequences of configurations, we need to express Gödel's β predicate in our FOL, so the configuration signature Σ needs to have a distinct sort $\mathbb{N}$ with constant symbols 0 and 1 and with binary operation symbols + and $\times$, which are interpreted in the configuration model $\mathcal{T}$ as the domain of natural numbers with corresponding constants and binary operations. We also need to assume that $\mathcal{T}$ can enumerate its own configurations. The weakest condition we were able to find in order to achieve that is to assume that Σ has an operation $\alpha : \mathit{Cfg} \to \mathbb{N}$ which is interpreted in $\mathcal{T}$ as an injective (one to one) function. To simplify writing, we deliberately make no distinction between operations in Σ and their interpretation in $\mathcal{T}$.

To summarize all the discussion above, in the remainder of of this section we will work under the following

Framework:
The reachability system $\mathcal{S}$ is
— non-empty;
— finite;
— well-defined; and
— finite branching.
The configuration signature Σ has
— a sort $\mathbb{N}$;
— constant symbols 0 and 1 of $\mathbb{N}$;
— binary operation symbols + and $\times$ on $\mathbb{N}$;
— an operation symbol $\alpha : \mathit{Cfg} \to \mathbb{N}$.
The configuration model $\mathcal{T}$ interprets
— $\mathbb{N}$ as the natural numbers;
— operation symbols on $\mathbb{N}$ as corresponding operations;
— $\alpha : \mathit{Cfg} \to \mathbb{N}$ as an injective function.

Recall that Var is a sort-wise infinite set of (first-order) variables, and that $\square$ is a special variable of sort Cfg such that $\square \notin \mathit{Var}$. Let $\mathit{Var}_{\mathcal{S}} \subset \mathit{Var}$ be the finite set of variables appearing free in any of the rules in $\mathcal{S}$. Notice that since $\square \notin \mathit{Var}$, it is also the case that $\square \notin \mathit{Var}_{\mathcal{S}}$. We let $c, c', c_0, \ldots, c_n$ be distinct variables of sort Cfg in $\mathit{Var} \setminus \mathit{Var}_{\mathcal{S}}$ (that is, these do not appear free in rules in $\mathcal{S}$). We also let $\gamma, \gamma', \gamma_0, \ldots, \gamma_n$ range over (not necessarily distinct) configurations in the model $\mathcal{T}$, that is, over elements in $\mathcal{T}_{\mathit{Cfg}}$, and let ρ, ρ' range over valuations $\mathit{Var} \to \mathcal{T}$.

Also recall that, by Proposition 1, matching logic formulae are a methodological fragment of the FOL theory of the model $\mathcal{T}$. For technical convenience, in this section we work with the FOL translations $\varphi^{\square}$ instead of the matching logic formulae φ. Moreover, since there is no possibility for confusion, we drop the $\square$ from $\varphi^{\square}$, and we use φ to denote the FOL translation. We mention that in all the formulae used in this section, $\square$ only occurs in the context $\square = t$, thus we stay inside the methodological fragment.

7.1 Gödelization of Configurations

We use Gödel's β predicate to encode facts about sequences of configurations in FOL (see [33] for an accessible introduction to Gödelization and the β predicate). The predicate β relies on the reminder of a when divided by b, namely a mod b. We have that $r = a \bmod b$ can be defined as

$$\exists d\ (b \times d \leq a \wedge b \times (d + 1) > a \wedge a = b \times d + r)$$

$\beta(a,\ b,\ i,\ x)$ is the predicate over natural numbers defined as

$$\beta(a,\ b,\ i,\ x) \equiv x = a \bmod 1 + (1 + i) \times b$$

Note that our assumptions about the configuration model $\mathcal{T}$ allow us to express Gödel's β predicate.

The main role of Gödel's β predicate is to encode, using conventional FOL, quantification over finite sequences of natural numbers. This is due to the following canonical property of β: if $u_0, \ldots, u_n$ is a sequence of natural numbers, then there exist natural numbers a and b such that for all i with $0 \leq i \leq n$ and x, we have $\models \beta(a,\ b,\ i,\ x) \leftrightarrow x = u_i$; in other words, a, b and i uniquely identify u_i. This allows us to take sentences $\exists u_0, \ldots, u_n\ \varphi$, where n is a given natural number and φ a FOL formula, and yield equivalent sentences $\exists a, b\ \overline{\varphi}$, where $\overline{\varphi}$ is a FOL formula obtained from φ by applying some systematic translation. As part of this translation, each atomic predicate p of φ is translated into

$$\exists u_{i_1}, \ldots, u_{i_k}\ (\beta(a,\ b,\ i_1,\ u_{i_1}) \wedge \cdots \wedge \beta(a,\ b,\ i_k,\ u_{i_k}) \wedge p)$$

where $u_{i_1}, \ldots, u_{i_k}$ are all the variables among $u_0, \ldots, u_n$ contained by p. The interesting cases are when $\exists u_0, \ldots, u_n\ \varphi$ occurs in contexts where n itself is existentially or universally quantified; in those cases, only a fixed (independent of n) subset of the variables $u_0, \ldots, u_n$ can occur in p, which means that the k above is fixed and independent of n, which means that $\overline{\varphi}$ is a correct FOL formula in those contexts. Thus, thanks to Gödel's β predicate, statements of the form $\exists u_0, \ldots, u_n\ \varphi$, which are not proper FOL formulae in contexts where n is a quantified variable, can be replaced with equivalent FOL formulae $\exists a, b\ \overline{\varphi}$.

It can be shown, although the proof is tedious and not necessary here, that the injectivity of $\alpha : \mathit{Cfg} \to \mathbb{N}$ allows us to adapt the result above to sequences of configurations in $\mathcal{T}$: sentences of the form $\exists c_0, \ldots, c_n\ \varphi$, where n is a given natural number and φ a FOL formula, can be systematically translated into equivalent sentences of the form $\exists a, b\, \overline{\varphi}$, where $\overline{\varphi}$ is a FOL formula replacing each atomic predicate p of φ containing variables $c_{i_1}, \ldots, c_{i_k}$ with

$$\exists c_{i_1}, \ldots, c_{i_k}\ (\beta(a,\ b,\ i_1,\ \alpha(c_{i_1})) \wedge \cdots \wedge \beta(a,\ b,\ i_k,\ \alpha(c_{i_k})) \wedge p)$$

The injectivity of α guarantees that different free occurrences of the same variable c_i in φ are correctly related in $\overline{\varphi}$. We only need the particular instance of this general result when φ expresses connectedness of $c_0, \ldots, c_n$ in $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$, and we prove that instance separately (Lemma 9).

7.2 Encoding Transition System Operations in FOL

We introduce the following definitions of: (1) the one step transition, (2) the transition sequence of length n (n is arbitrary but fixed), (3) the infinite transition sequence, (4) the configurations that reach φ, and (5) the configurations that reach φ in at least one step. Except for the first definition of the *step* predicate below, which is a proper FOL formula because $\mathcal{S}$ is finite, the definitions below quantify over sequences of configurations $c_0, \ldots, c_n$, so they are not (yet) FOL formulae. Lemma 9 shows how to Gödelize such sequences and thus tells us that we can express all these predicates in FOL. For simplicity, the first three definitions do not allow $\square$ as a parameter. Thus, we cannot directly use $path_n(\square,\ c')$ and $step(\square,\ c'')$ in the fourth and fifth definitions, respectively. Instead, we introduce an existentially quantified variable c which is constrained to be equal to $\square$.

$$\begin{aligned}
step(c,\ c') &\equiv \bigvee_{left \Rightarrow right \in \mathcal{S}} \exists Var_{\mathcal{S}}\ (left[c/\square] \wedge right[c'/\square]) \\
path_n(c,\ c') &\equiv \exists c_0 c_1 \ldots c_n\ (c = c_0 \wedge c' = c_n \\
&\qquad \wedge \bigwedge_{1 \leq i \leq n} step(c_{i-1},\ c_i)) \\
infinite(c) &\equiv \forall n\, \exists c'\ path_n(c,\ c') \\
coreach(\varphi) &\equiv \exists n\, \exists c\, \exists c'\ (c = \square \wedge \varphi[c'/\square] \wedge path_n(c,\ c')) \\
coreach^{+}(\varphi) &\equiv \exists c''\ (\exists c\ (\square = c \wedge step(c,\ c'')) \\
&\qquad \wedge coreach(\varphi)[c''/\square])
\end{aligned}$$

The following lemmas state that the above definitions actually have the semantic properties their names suggest. Recall that $c, c', c_0, \ldots, c_n$ are distinct variables of sort Cfg in $Var \setminus Var_{\mathcal{S}}$.

Lemma 7. *$\rho \models step(c,\ c')$ iff $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$.*

Lemma 8. *Let n be a natural number. Then $\rho \models path_n(c,\ c')$ iff $\rho(c) \Rightarrow^{n\mathcal{T}}_{\mathcal{S}} \rho(c')$.*

Now, using Gödel's β predicate and the general schema described in Section 7.1, we can define a transition sequence of length n (n is arbitrary but fixed) in FOL using only a fixed number of quantifiers as shown in Figure 9. Formally, we have the following relationship:

Lemma 9. $\models path_n(c,\ c') \leftrightarrow \overline{path}_n(c,\ c')$.

Consequently, we can use $\overline{path}_n(c,\ c')$, the alternative equivalent definition of $path_n(c,\ c')$, to express $infinite(c)$, $coreach(\varphi)$ and $coreach^{+}(\varphi)$ in FOL. Since our relative completeness proof only uses $step(c,\ c')$, $infinite(c)$, $coreach(\varphi)$ and $coreach^{+}(\varphi)$ besides other FOL formulae over the signature Σ, we can conclude that all the formulae used in our proof are FOL formulae. For notational simplicity, we however prefer to continue to work with $path$ instead of $\overline{path}$.

Since $\mathcal{S}$ is finite branching, a configuration in $\mathcal{T}_{Cfg}$ does not terminate in $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ if and only if it yields finite paths of any length. Formally, we can prove the following:

Lemma 10. *$\rho \models infinite(c)$ iff $\rho(c)$ does not terminate in $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$.*

Lemma 10 implies the following valid formula (of $\mathcal{T}$):

Lemma 11. $\models infinite(c) \rightarrow \exists c'(step(c,\ c') \wedge infinite(c'))$.

Lemma 8 implies the following property of $coreach(\varphi)$:

Lemma 12. *$(\gamma, \rho) \models coreach(\varphi)$ iff there exists some γ' with $(\gamma', \rho) \models \varphi$ and $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$.*

The following result formally establishes the expected relationship between φ, $coreach(\varphi)$, and $coreach^{+}(\varphi)$:

Lemma 13. $\models coreach(\varphi) \leftrightarrow \varphi \vee coreach^{+}(\varphi)$.

7.3 Encoding Semantic Validity in FOL

Here we show that the semantic validity of matching logic reachability rules can be framed as FOL validity.

Proposition 3. *If $\mathcal{S} \models \varphi \Rightarrow \varphi'$ then the FOL validity*

$$\models \varphi \rightarrow \exists c\ (\square = c \wedge infinite(c)) \vee coreach(\varphi')$$

holds in $\mathcal{T}$.

Note that the proposition above, while reducing semantic validity of our reachability sequents to FOL validity, does not yet prove the desired relative completeness result. As an analogy, consider the relative completeness of Hoare logic for some simple language. Similarly, one can reduce the semantic validity of a Hoare triple $\{\psi\}\,\texttt{code}\,\{\psi'\}$ to FOL validity, but that does *not* directly imply that $\{\psi\}\,\texttt{code}\,\{\psi'\}$ is derivable with the Hoare logic proof system. One still has to construct a proof derivation for $\{\psi\}\,\texttt{code}\,\{\psi'\}$ using the available proof system, which is the hard part of the relative completeness result for Hoare logic. Similarly, we still have to construct a derivation for our reachability rule $\varphi \Rightarrow \varphi'$ using the available matching logic proof system in Figure 7.

7.4 Relative Completeness

We are now ready to start using the proof system to derive a semantically valid reachability rule. We do so by proving several helping lemmas.

The next lemma states that if a logical step (in terms of the *step* predicate) is possible from the current configuration to a next configuration, then that next configuration is provably reachable without Reflexivity. The result may sound obvious, but, however, it requires the well-definedness of $\mathcal{S}$. For

$$\overline{path}_n(c,\ c') \ \equiv\ \exists a \exists b\, (\exists c_0\, (\beta(a,\ b,\ 0,\ \alpha(c_0)) \wedge c = c_0) \wedge \exists c_n\, (\beta(a,\ b,\ n,\ \alpha(c_n)) \wedge c' = c_n)$$
$$\wedge \forall i\, (1 \le i \wedge i \le n \to \exists c_{i-1} \exists c_i\, (\beta(a,\ b,\ i-1,\ \alpha(c_{i-1})) \wedge \beta(a,\ b,\ i,\ \alpha(c_i)) \wedge step(c_{i-1},\ c_i))))$$

Figure 9. FOL definition of a transition sequence

example, if $\mathcal{S}$ consists of a rule of the form $\pi \Rightarrow \pi_1 \vee \pi_2$ then the rule below cannot be derivable because, if it were, then by the Substitution derived rule (Lemma 4), it would also be derivable when c' is substituted with π_1; however, in that case the rule becomes $\pi \Rightarrow \pi_1$, which is not derivable (see the discussion in the preamble of Section 7).

Lemma 14. $\mathcal{S} \vdash \exists c\, (\square = c \wedge step(c,\ c')) \Rightarrow^+ \square = c'$.

The next lemma states that the reachability rule whose both patterns specify non-termination is derivable without Reflexivity. Intuitively, that is because the *step* predicate will relate a non-terminating configuration with at least one next non-terminating configuration. This lemma enables the use of the Circularity rule later (in Lemma 17).

Lemma 15. $\mathcal{S} \vdash \exists c (\square = c \wedge infinite(c)) \Rightarrow^+ \exists c (\square = c \wedge infinite(c))$

Lemma 16. *If* $\mathcal{S} \cup \mathcal{A} \vdash coreach^+(\varphi) \Rightarrow \varphi$, *then*

$$\mathcal{S} \cup \mathcal{A} \vdash coreach(\varphi) \Rightarrow \varphi$$

The following result is critical for the completeness theorem: it says that the reachability rule from the pattern specifying either non-terminating configurations or configurations reaching φ to φ is derivable:

Lemma 17. $\mathcal{S} \vdash \exists c\, (\square = c \wedge infinite(c)) \vee coreach(\varphi) \Rightarrow \varphi$.

Finally, the relative completeness result now follows from all the above lemmas. Note how the configuration model is being used, via Proposition 3, as an oracle to answer the semantic reachability question formulated as a FOL sentence.

Theorem 2. *If* $\mathcal{S} \models \varphi \Rightarrow \varphi'$ *then* $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$.

Proof. Assume that $\mathcal{S} \models \varphi \Rightarrow \varphi'$. Then, by Proposition 3, we have that

$$\models \varphi \to \exists c\, (\square = c \wedge infinite(c)) \vee coreach(\varphi')$$

Further, by Lemma 17, we have that

$$\mathcal{S} \vdash \exists c\, (\square = c \wedge infinite(c)) \vee coreach(\varphi) \Rightarrow \varphi$$

Then the theorem follows by Consequence. □

8. Implementation and Evaluation

Here we discuss our MatchC implementation of the proof system in Figure 7. 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 reachability 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). MatchC takes as inputs code fragments written in a C fragment and (user provided) specifications for functions and loops, and automatically checks that the code respect the specifications (without user interaction or additional annotations, like ghost variables or hints).

As discussed in Section 5, general matching logic specifications are reachability rules between formulae. As seen in Section 3, our tool handles specifications of the form:

$$\exists X (\pi \wedge \psi) \Rightarrow \exists X' (\pi' \wedge \psi')$$

where: π and π' are basic patterns; ψ and ψ' are patternless FOL formulae; X and X' are sets of first-order variables; π contains the cell $\langle \texttt{code}\ \cdots\rangle_{\mathsf{k}}$ and π' contains the cell $\langle \texttt{code'}\ \cdots\rangle_{\mathsf{k}}$; and `code'` is either "·" or the **`return`** statement. For now, MatchC only supports (partial correctness) rules summarizing the behavior of functions or loops. An invariant $\exists X (\pi \wedge \psi)$ for **`while`**`(C)S` is just syntactic sugar for a reachability rule. For clarity, we 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 $v_{\mathtt{x}}$, we associate with the loop the following rule:

$$\exists X (\pi \wedge \psi) \Rightarrow \exists X (\pi' \wedge \psi \wedge v_{\mathtt{x}} = 0)$$

where π' is obtained from π by replacing $\langle \textbf{while}\texttt{(x)S}\ \cdots\rangle_{\mathsf{k}}$ with $\langle \cdot\ \cdots\rangle_{\mathsf{k}}$, i.e., dropping the loop. The above rule summarizes the loop. Section 6.2 discusses the reachability rule associated to the loop invariant of `reverseList` in Figure 2.

We define the operational semantics of the C fragment in the $\mathbb{K}$ framework [26] as a set of reachability rules $\mathcal{S}$ over the configuration in Figure 6. Let us discuss in more detail how to use the proof system in Figure 7 and the derived proof rules in Section 6.1 to derive symbolic execution using $\mathcal{S}$ as axioms. For example, consider the reachability rule in the Example 1. First, we look at one step of symbolic execution, and then we consider formula abstraction.

Figure 10 formally derives the execution of the assignment `y = x->next` (assuming that `x->next` evaluates to some value q). The `HEAP` macro stands for the cell $\langle \mathsf{list}(?\mathsf{p})(?\mathsf{B}),\ ?\mathsf{x} \mapsto [a, q],\ \mathsf{list}(q)(\beta),\ \mathsf{H}\rangle_{\mathsf{heap}}$. We give below the reachability rule associated to the $\mathbb{K}$ semantics of assignment (in C assignment is an expression):

$$\langle\langle \mathsf{X}\ =\ \mathsf{V} \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\ \langle \mathsf{X} \mapsto \mathsf{P},\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathsf{C}\rangle_{\mathsf{cfg}}$$
$$\Rightarrow\quad \langle\langle \mathsf{V} \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\ \langle \mathsf{X} \mapsto \mathsf{V},\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathsf{C}\rangle_{\mathsf{cfg}}$$

Like refocussing [7], $\mathbb{K}$ flattens computations in a stack-like structure whose tasks are separated by $\curvearrowright$ (read "followed by"), whose intuition is that its left argument is first evaluated and then its value is passed to its right argument, which inserts

$$\dfrac{\dfrac{\dfrac{\cdot}{\mathcal{S} \vdash \langle\langle \mathsf{X} \;=\; \mathsf{V} \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathsf{X} \mapsto \mathsf{P},\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathsf{C}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathsf{V} \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathsf{X} \mapsto \mathsf{V},\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathsf{C}\rangle_{\mathsf{cfg}}}\ \text{Axiom}}{\mathcal{S} \vdash \langle\langle \mathtt{y} \;=\; q \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathtt{x} \mapsto ?\mathsf{x},\ \mathtt{p} \mapsto ?\mathsf{p},\ \mathtt{y} \mapsto ?\mathsf{y},\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathtt{HEAP}\ \mathsf{C}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathtt{x} \mapsto ?\mathsf{x},\ \mathtt{p} \mapsto ?\mathsf{p},\ \mathtt{y} \mapsto q,\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathtt{HEAP}\ \mathsf{C}\rangle_{\mathsf{cfg}}}\ \text{Subst}}{\mathcal{S} \vdash \langle\langle \mathtt{y} \;=\; q \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathtt{x} \mapsto ?\mathsf{x},\ \mathtt{p} \mapsto ?\mathsf{p},\ \mathtt{y} \mapsto ?\mathsf{y},\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathtt{HEAP}\ \mathsf{C}\rangle_{\mathsf{cfg}} \wedge \psi \Rightarrow \langle\langle \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathtt{x} \mapsto ?\mathsf{x},\ \mathtt{p} \mapsto ?\mathsf{p},\ \mathtt{y} \mapsto q,\ \mathsf{E}\rangle_{\mathsf{env}}\ \mathtt{HEAP}\ \mathsf{C}\rangle_{\mathsf{cfg}} \wedge \psi}\ \text{LF}$$

Figure 10. Formal derivation of symbolic assignment

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{S} \vdash \langle\langle {*}(?\mathsf{x} +_{Int} 1) \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathsf{list}(?\mathsf{p})(?\mathsf{B}), ?\mathsf{x} \mapsto [a, q],\ \mathsf{list}(q)(\beta),\ \mathsf{H}\rangle_{\mathsf{heap}}\ \mathtt{ENV}\ \mathsf{C}\rangle_{\mathsf{cfg}} \wedge \psi' \wedge ?\mathsf{x} \neq 0 \wedge ?\mathsf{C} = [a]@\beta \Rightarrow \ldots}}{\mathcal{S} \vdash \exists a \exists q\, (\langle\langle {*}(?\mathsf{x} +_{Int} 1) \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathsf{list}(?\mathsf{p})(?\mathsf{B}), ?\mathsf{x} \mapsto [a, q],\ \mathsf{list}(q)(\beta),\ \mathsf{H}\rangle_{\mathsf{heap}}\ \mathtt{ENV}\ \mathsf{C}\rangle_{\mathsf{cfg}} \wedge \psi' \wedge ?\mathsf{x} \neq 0 \wedge ?\mathsf{C} = [a]@\beta) \Rightarrow \ldots}\ \text{Abs}}{\mathcal{S} \vdash \langle\langle {*}(?\mathsf{x} +_{Int} 1) \curvearrowright \mathsf{K}\rangle_{\mathsf{k}}\, \langle \mathsf{list}(?\mathsf{p})(?\mathsf{B}),\ \mathsf{list}(?\mathsf{x})(?\mathsf{C}),\ \mathsf{H}\rangle_{\mathsf{heap}}\ \mathtt{ENV}\ \mathsf{C}\rangle_{\mathsf{cfg}} \wedge \psi' \wedge ?\mathsf{x} \neq 0 \Rightarrow \ldots}\ \text{Cons}$$

Figure 11. Use of Consequence and Abstraction to reduce a more abstract configuration (bottom) to a more concrete one (up).

it in the appropriate place, marked by ■. Let $\theta : Var \to \mathcal{T}_{\Sigma}(Var)$ be a substitution mapping the free variables in the rule above as follows:

$\theta(\mathsf{X}) = \mathtt{y}$
$\theta(\mathsf{V}) = q$
$\theta(\mathsf{K}) =$ ■`; x->next = p; p = x; x = y;`
$\theta(\mathsf{P}) = ?\mathsf{y}$
$\theta(\mathsf{E}) = \mathtt{x} \mapsto ?\mathsf{x},\ \mathtt{p} \mapsto ?\mathsf{p},\ \mathsf{E}$
$\theta(\mathsf{C}) = \langle \mathsf{list}(?\mathsf{p})(?\mathsf{B}), ?\mathsf{x} \mapsto [a, q],\ \mathsf{list}(q)(\beta),\ \mathsf{H}\rangle_{\mathsf{heap}}\ \mathsf{C}$

and let ψ be the patternless first-order formula $?\mathsf{C} = [a]@\beta \wedge \mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C} \wedge ?\mathsf{x} \neq 0$. Then, we can symbolically execute the assignment by applying the Logic Framing proof rule with the frame ψ, the Substitution derived proof rule with the substitution θ above and the Axiom proof rule with the rule above, as shown in Figure 10. The use of this sequence of Axiom, Substitution and Logic Framing is generic and entirely automatic. Further, the Transitivity proof rule allows the chaining of such sequences. The Case Analysis proof rule allows for splitting on constructors such as **`if`** when their conditions evaluate to symbolic values.

Next, we consider an issue that arises due to formula abstraction: the configuration can be too abstract for the semantic rule to apply via Axiom. In such a case, the Consequence and Abstraction proof rules can reduce proving a reachability property about a more abstract configuration, which is not matched by any semantic rules, to a property about a more concrete configuration. Figure 11 shows such an example caused by the symbolic memory read from the location ${*}(?\mathsf{x} +_{Int} 1)$, which occurs as part of the evaluation of `x->next` in the context in Figure 8. The macro `ENV` stands for $\langle \mathtt{x} \mapsto ?\mathsf{x},\ \mathtt{p} \mapsto ?\mathsf{p},\ \mathtt{y} \mapsto ?\mathsf{y},\ \mathsf{E}\rangle_{\mathsf{env}}$, while ψ' is $\mathsf{A} = \mathsf{rev}(?\mathsf{B})@?\mathsf{C}$. We use Consequence with the list abstraction axiom in Section 4.2 and the fact that $?\mathsf{x} \neq 0$ followed by Abstraction with $\{a,\ q\}$ and then we can symbolically evaluate ${*}(?\mathsf{x} +_{Int} 1)$ in a configuration in which $?\mathsf{x} +_{Int} 1$ is explicitly mentioned in the heap. In general, we can use the first seven rules of the proof system (the proof system without Circularity) to derive the symbolic execution of a linear segment of code (code without circular behaviours).

Let C be the set of reachability rules specifying all the user provided program properties. C contains one candidate rule for each function and one candidate rule for each loop. MatchC derives the rules in C by applying the proof rules in Figure 7 and the derived proof rules in Section 6.1 according to certain heuristics. It begins by applying Circularity followed by Transitivity for each rule in C and reduces the tasks to deriving individual sequents of the form $\mathcal{A} \cup C \vdash \exists X(\pi \wedge \psi) \Rightarrow \exists X'(\pi' \wedge \psi')$. To prove each such rule, the tool symbolically executes the code in the left-hand-side formula using axioms from $\mathcal{A} \cup C$ (like in the example above), and then checks that the formulae obtained after the execution imply the right-hand-side formula. Recall that the code of the right-hand-side is either "$\cdot$" or **`return`**, so we know how the symbolic execution should terminate.

For each left-hand-side there may be multiple execution paths, generated by splits via Case Analysis on constructors like **`if`** or on disjunctions existent in the specifications or introduced by abstraction axioms or domain reasoning. Similarly, when the configuration is too abstract for any rule in $\mathcal{A} \cup C$ to apply, the tool uses abstraction axioms to obtain a more concrete configuration if certain triggers are met; in the example above, the memory access on the head of the list triggered the unrolling. As an optimisation, when a formula can be reduced with rules from both $\mathcal{S}$ and C, the verifier only uses the rules from C. In particular, only a loop without a specified invariant is unrolled, and only the body of a function without a rule specification is executed. Another heuristic is that 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 list is not null results in an automatic list unrolling). MatchC is therefore sound but incomplete w.r.t. the reachability proof system.

The symbolic execution is also implemented in $\mathbb{K}$, as a set of rules which are added to the original set of semantic rules. Checking of matching logic formulae implication (required for Consequence) is implemented in Maude [6]. 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 abstraction 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 [2] and Z3 [8]. MatchC comes with a library of ~100 mathematical domain operators (like rev, in) and pattern abstractions (like list), together with their axioms and useful lemmas (see Section 4.2). It currently provides support for reasoning about lists, trees, queues and graphs.

Figure 12 summarises the results of our experiments (# paths column gives the number of symbolic execution paths analysed). 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 2 , 3 and 5. 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 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. The Schorr-Waite algorithm presents considerable verification challenges [17, 18]. We formally verified the algorithm itself, 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. Given the complexity of the specifications and the level of automation, the average time per program (below one second) is low and not a matter of concern. The experiments were conducted on a quad-core, 2.2GHz, 4GB machine running Linux.

9. Conclusion and Future Work

This paper presented an eight-rule proof system for reachability, parametric in an operational semantics of the target programing language. The proof system was proved partially correct and relatively complete. At our knowledge, this is the first language-independent proof system with these properties. Previous proof systems, such as those corresponding to Hoare/separation/dynamic logics and extensions of them, are language-specific and need to be proved sound with respect to another semantics of the same language.

With the help of Stefan Ciobaca and Brandon Moore, we have started developing in Coq a certifiable program verification framework based on the presented proof system.

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 12. Results of MatchC program verification

Our progress can be seen at `http://fsl.cs.uiuc.edu/ml`. Our objective is to develop language-independent tactics that allow for compact and intuitive proofs of program correctness, as well as for generation of certifiable proof objects relying only on the operational semantics of the target language and requiring no low-level inductive proofs about the transition system associated to the operational semantics, and no language-specific lemmas. We also intend to generalize our efficient MATCHC prover into a generic one, and connect it with the Coq-based framework, to achieve both efficient and certifiable verification based on the presented proof system.

Acknowledgements: We thank the members of the $\mathbb{K}$ team (`http://k-framework.org`) and the anonymous reviewers for their valuable comments on a previous version of this paper. The work in this paper was supported in part by NSA contract H98230-10-C-0294, by NSF grant CCF-0916893 and by (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

[1] A. W. Appel. Verified software toolchain. In *ESOP*, volume 6602 of *LNCS*, pages 1–17, 2011.

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

[3] G. Berry and G. Boudol. The chemical abstract machine. *Th. Comp. Sci.*, 96(1):217–248, 1992.

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

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

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

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

[8] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[9] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302, 2006.

[10] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[11] 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.

[12] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT, 2009.

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

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

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

[16] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, volume 4960 of *LNCS*, pages 353–367, 2008.

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

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

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

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

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

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

[23] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

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

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

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

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

[28] G. Rosu and A. Stefanescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP (2)*, volume 7392 of *LNCS*, pages 351–363, 2012.

[29] G. Rosu and A. Stefanescu. From Hoare logic to matching logic. In *FM*, To appear, 2012.

[30] G. Rosu and A. Stefanescu. Checking reachability using matching logic. Technical Report `http://hdl.handle.net/2142/33771`, Univ. of Illinois, Aug. 2012.

[31] G. Rosu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, volume 6486 of *LNCS*, pages 142–162, 2010.

[32] 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.

[33] G. Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

[34] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. & Computation*, 115(1):38–94, 1994.